Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт
Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О. В. Непомнящий
подпись инициалы, фамилия
«___» ____ 2019 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 Информатика и вычислительная техника
код – наименование направления

Параллельная программа для вычисления функций роста бернсайдовых двупорожденных групп на гибридных вычислительных системах
тема

Руководитель	_____	доцент, канд. техн. наук	Д. А. Кузьмин
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	_____		С. А. Тарасов
	подпись, дата		инициалы, фамилия
Нормоконтролер	_____	доцент, канд. техн. наук	В. И. Иванов
	подпись, дата	должность, ученая степень	инициалы, фамилия

Красноярск 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Многие фундаментальные работы в различных отраслях науки долгое время после их публикации не находят прикладного применения. Особенно актуальна эта проблема для математических открытий в силу чрезвычайной абстрактности математических объектов и концепций.

Еще в XIX веке английским математиком Артуром Кэли было дано определение графам со специфическими свойствами, которые в дальнейшем получили его имя – графы Кэли; и только относительно недавно оценено их практическое значение.

Граф Кэли – это граф, построенный по алгебраической группе с выделенной системой образующих. Правило построения графа Кэли произвольной группы с выделенной системой образующих задано следующим образом: каждому элементу группы ставится в соответствие одна вершина графа Кэли, причем ребрами между собой соединяются любые две вершины графа такие, что первая из них соответствует некоторому произвольному элементу группы, а вторая – элементу группы, полученному домножением первого элемента на некоторый элемент системы образующих.

Помимо фундаментальной ценности, Графы Кэли имеют большую практическую ценность, например, в следующих областях: проектировании интегральных полупроводниковых схем, организации компьютерных сетей [1, с. 216-223], криптографии, информационном кодировании, проектировании суперкомпьютеров [1, с. 216-223; 2, с. 337-357], проектировании баз данных.

В настоящее время на фоне роста актуальности проблемы эффекта больцмановской тирании, наиболее перспективно выглядит использование графов Кэли при проектировании топологий многопроцессорных вычислительных систем (МВС) [3; 4 с. 138].

Если представить процессорную матрицу МВС в виде графа, то процессоры в графе будут являться узлами, а ребра графа – физическими соединениями между процессорами. Характеристики такого графа (связность и диаметр) отражают характеристики соответствующей МВС [3]. Таким образом, решение задачи нахождения графа с наименьшим диаметром и наименьшей степенью вершин при фиксированном числе вершин одновременно является решением задачи поиска эффективной топологии МВС. Именно графы Кэли обладают указанными свойствами. Так, например, получили распространение такие топологии МВС как «кольцо», «гиперкуб» и «тор», которые являются графами Кэли [3].

Процессорные матрицы МВС спроектированные на основе структуры графов Кэли имеют наилучшие архитектурно-зависимые характеристики, главная из которых – минимальная физически-обусловленная латентность доступа от одного процессорного элемента к другому [3].

Помимо непосредственного использования структуры графов Кэли как основы каких-либо перспективных топологий, эти графы находят применение в

качестве инструмента решения класса комбинаторных и оптимизационных задач, связанных с перестановками – перестановочных задач [4, с. 147-148]. К таким задачам относят, например, перестановочные головоломки – кубик Рубика и Хайнойскую башню, а также проблему оптимального использования ограниченных ресурсов и проблему маршрутизации в компьютерных сетях.

Любую перстановочную задачу можно описать с помощью группы подстановок, которую, в свою очередь, можно представить в виде графа Кэли. В этом случае граф Кэли будет содержать все возможные решения задачи, а также некоторую другую полезную информацию.

В качестве примера рассмотрим граф Кэли кубика Рубика. Каждой вершине Графа Кэли группы кубика Рубика соответствует определенная конфигурация кубика, ребрам – допустимые переходы между конфигурациями (поворот грани на угол 90° или 180°). Такой граф содержит все возможные решения головоломки для всех возможных пар начальных и конечных конфигураций. Кроме того, диаметр такого графа равен минимальному числу элементарных ходов, последовательное выполнение которых гарантированно приводит к решению головоломки для всех возможных пар начальных и конечных конфигураций кубика. Это число принято называть числом Бога. Для классического кубика Рубика число Бога равно 20, то есть диаметр соответствующего графа равен 20. Эти свойства графа Кэли действительны и для других перестановочных головоломок, но в общем случае число Бога головоломки равно не диаметру ее графа Кэли, а эксцентриситету вершины, соответствующей конечной конфигурации. В случае кубика Рубика диаметр его графа Кэли равен эксцентриситету любой из вершин. Не трудно представить, что аналогично графы Кэли возможно применять при решении более серьезных задач.

Сегодня особый интерес представляют графы Кэли бернсайдовых двупорожденных групп периода 5 – B(2, 5) [3].

Диаметр графа Кэли можно вычислить с помощью функции роста соответствующей этому графу группы.

К сожалению, вычисление функции роста большой конечной группы является хотя и разрешимой, но весьма сложной проблемой – например, вычислить функцию роста группы $B_0(2, 5)$ порядка выше 5^{17} в настоящее время не удается, так как существующие компьютерные реализации алгоритмов, предназначенных для решения этой задачи, требуют неприемлемо больших вычислительных ресурсов – процессорного времени и объема оперативной памяти [3].

Относительно недавно математиком А. А. Кузнецовым [3] был предложен оптимизированный алгоритм для вычисления функций роста групп $B_0(2, 5)$ и диаметра соответствующего графа Кэли, а также была доказана корректность этого алгоритма. Алгоритм получил название «модифицированный алгоритм A-I». Кроме того, алгоритм был реализован для SMP-архитектуры в соответствии со стандартом OpenMP [3].

К сожалению, в настоящее время не существует реализаций модифицированного алгоритма A-I для MPP-систем, а также GPGPU-систем, поэтому остаются малоизученными архитектурно-зависимые свойства этого алгоритма. Целью настоящей бакалаврской работы является частичное восполнение указанного пробела, а именно реализация модифицированного алгоритма A-I для GPGPU-системы.

1 Постановка задачи

1.1 Цель

Главная цель настоящей работы – реализовать параллельную модификацию алгоритма A-I, предназначенного для вычисления функций роста бернсайдовых двупорожденных групп степени 5 B(2, 5), в парадигме GPGPU, а также дать оценку эффективности реализации в соответствие с нижеописанными критериями.

1.2 Критерии оценки

Для оценки GPGPU-реализации модифицированного параллельного алгоритма A-I использовать следующие основные критерии: реальная вычислительная пространственная сложность, реальная вычислительная временная сложность, масштабируемость.

2 Обзор алгоритмов

Модифицированный параллельный алгоритм A-I основан на базовом последовательном алгоритме A-I. В настоящей главе дается неформальное описание этих алгоритмов.

2.1 Обзор базового алгоритма A-I

Базовый алгоритм A-I [3] за конечное число шагов вычисляет функцию роста любой конечной бернсайдовой группы G, заданной фиксированным порождающим множеством X.

Неформальное описание A-I имеет следующий вид – листинг 1, где X – порождающее множество конечной бернсайдовой группы G, F(s) – ее функция роста, $D_X(G)$ – диаметр графа Кэли группы G, K_s – шар радиуса s группы G, P – множество уникальных элементов группы G, вычисленных на шаге 5.

Базовый алгоритм A-I можно разделить на три основных повторяющихся этапа: 1 – вычисление элементов шара K_s, вычисление разности P шаров K_s и K_{s-1}, вычисление мощности полученной разности. Мощность разности P и является значением функции роста от радиуса s.

Критерием останова алгоритма является вырождение функции роста – $F(s) = 0$.

Листинг 1 – Базовый алгоритм A-I [3]

1 Вход: X = {x1, x2, ..., xm} – порождающее множество G
2 Выход: функция роста F(s) группы G, диаметр $D_X(G)$ диаметр графа Кэли группы G
3 1: s := 0, F(0) := 1, K_0 := {e}, P := K_0
4 2: s := s + 1
5 3: $K_s := K_{s-1}$
6 4: Для всех x ∈ X и p ∈ P
7 5: g = x·p (или g = p·x)
8 6: Если g ∉ K_s,
9 то $K_s := K_s \cup \{g\}$
10 7: $P := K_s - K_{s-1}$
11 8: F(s) := |P|
12 9: Если F(s) > 0,
13 то переход в п.2,
14 10: иначе
15 s_0 := s − 1, переход в п.11
16 11: $D_X(G) := s_0$, выход

Анализ алгоритма A-I показал, что его вычислительная сложность зависит от мощности порождающего множества следующим образом [3]: если $|X| << |G|$, то $T(|G|) \in \Theta(|G|^2)$; если $|X| \sim |G|$, то $T(|G|) \in O(|G|^3)$; где T – вычислительная сложность A-I, а Θ – одновременно верхняя и нижняя оценки вычислительной сложности A-I. Таким образом, алгоритм A-I является NP-

трудным [3], и в наихудшем случае количество элементарных операций, которые необходимо выполнить для решения указанной задачи, представляет собой экспоненциальную функцию от |X|.

Помимо предполагаемой временной вычислительной сложности данного алгоритма, велика и его предполагаемая пространственная сложность, так как алгоритм требует хранения большого числа элементов исследуемой группы, количество которых также увеличивается экспоненциально при увеличении |X| (экспоненциальная функция от |X|).

2.2 Обзор модифицированного алгоритма A-I

Важно отметить узкие места алгоритма из листинга 1. Если порядок группы G достаточно большой, то наиболее критическими участками алгоритма A-I будут пункты 5 и 6, то есть умножение элементов группы и проверка, позволяющая определить, встречался ли ранее элемент.

Модифицированный алгоритм A-I [3] использует другой подход к умножению элементов в группах вместо собирательного процесса используются полиномы Холла. Вычислительные эксперименты показали [3], что полиномы Холла позволяют значительно быстрее собирательного процесса (по крайней мере на порядок) умножать элементы в группах. Таким образом был модифицирован пункт 5 базового алгоритма.

Для оптимизации пункта 6 исходного алгоритма A-I введен булев вектор V размерности 5^k, все элементы которого изначально равны 0; каждому элементу g группы G присвоен порядковый номер n_g; значение элемента вектора V_{ng} показывает принадлежность элемента g шару K_s: если $V_{ng} = 1$, то элемент $g \in K_s$; если $V_{ng} = 0$, то элемент $g \notin K_s$. Такой подход позволил свести к линейной вычислительной сложности поиск элемента g во множестве K_s.

Благодаря введенным модификациям алгоритма A-I стало возможным эффективно распараллелить вычисления соответствующие пунктам 5 и 6 оригинального алгоритма.

Модифицированный алгоритм A-I имеет следующий вид – листинг 2.

Листинг 2 – Модифицированный алгоритм A-I [1]

```
1   Вход: X = {x1, x2, ..., xm} – порождающее множество G
2   Выход: функция роста F(s) группы G, диаметр D_X(G)  диаметр графа Кэли группы G
3   1: s := 0, F(0) := 1, K_0 := {e}, P := K_0, V = (0, 0, …, 0)
4   2: s := s + 1
5   3: K_s := K_{s-1}
6   4: Для всех x ∈ X и p ∈ P
7   5: g = x · p (или g = p · x)
8   6: Если V_ng = 0,
9           то V_ng := 1 и K_s := K_s ∪ {g}.
10  7: P := K_s − K_{s-1}
11  8: F(s) := |P|
12  9: Если F(s) > 0,
```

то переход в п.2,

10: иначе

$s_0 := s - 1$, переход в п.11

11: $D_X(G) := s_0$, выход

Модифицированный алгоритм A-I имеет следующую оценку вычислительной сложности [3]: $T(|B_k|)$, $T(|B_k|) \in \Theta(|B_k|)$, где B_k – бернсайдова группа степени k.

3 Обзор аппаратного обеспечения

При создании высокопроизводительных программ, к сожалению, не возможно в достаточной степени абстрагироваться от аппаратного обеспечения, поэтому в настоящей главе приведен краткий обзор аппаратной платформы, на которой предполагается апробация алгоритма.

Для решения поставленной задачи Центром высокопроизводительных вычислений СФУ было выделено оборудование – компьютер IBM System x DataPlex dx360 M4, который имеет следующие значимые для решаемой задачи параметры: графический процессор – NVIDIA Tesla K20m, 2 центральных процессора – Intel(R) Xeon(R) CPU E5-2630L с тактовой частотой 2.00GHz и 6-ю ядрами (по 2 потока на ядро), оперативная память DDR3 общим объемом 188 Гб, технология соединения центрального и графического процессоров – PCI Express 2.0.

Наиболее интересны в контексте поставленной задачи характеристики графического ускорителя – таблица 1.

Таблица 1 – Характеристики акселератора NVIDIA Tesla K20m

Характеристика	Значение
Аппаратная архитектура	Kepler
Вычислительная способность	3.5
Количество CUDA-ядер	2496
Количество потоковых мультипроцессоров (SMX)	13
Объем глобальной памяти	4742 Мб
Объем константной памяти	65536 байт
Объем разделяемой памяти на блок	49152 байт
Количество 32-разрядных регистров на блок	65536
Размер варпа	32
Максимальное количество потоков на мультипроцессор	2048
Максимальное количество потоков на блок	1024
Максимальный размер блока (x, y, z)	(1024, 1024, 64) нитей
Максимальный размер сетки (x, y, z)	(2147483647, 65535, 65535) блоков
Ограничение на число запускаемых блоков	нет
Поддержка универсального адресного пространства (UVA)	есть
Поддержка прямого доступа к закрепленной памяти хоста (pinned memory)	есть
Тактовая частота мультипроцессоров	706 МГц
Тактовая частота глобальной памяти	2600 МГц

Окончание таблицы 1

Характеристика	Значение
Разрядность шины памяти	320 бит
Поддержка интеграции внешней памяти	нет
Поддержка прямого доступа к памяти другого GPU (peer-to-peer)	есть
Версия PCI Express	2.0
Количество линий PCI Express	16

В нижеизложенных тезисах использованы семантически схожие понятия – глобальная оперативная память [5] (или просто глобальная память) и внешняя оперативная память (или просто оперативная память). Стоит обратить внимание на то, что в данном контексте эти понятия не следует обобщать: глобальная память – один из видов памяти GPU, который на физическом уровне реализован с помощью микросхем DRAM-памяти, расположенных непосредственно на плате акселератора; оперативная память – оперативная память в традиционном понимании, для данной вычислительной системы реализованная на физическом уровне с помощью DDRAM-модулей, установленных в слоты интерфейса PCI Express на материнской плате компьютера.

Из таблицы 1 можно выделить основные характеристики, которые в наибольшей степени повлияют на программные решения, принятые при реализации модифицированного алгоритма A-I.

Во-первых, необходимо обратить внимание на объем глобальной памяти акселератора Tesla K20m, который в данном случае равен приблизительно 5 Гб. Эта характеристика является критически важной для видеокарт NVIDIA, так как все целевые обрабатываемые акселератором данные неминуемо помещаются в глобальную память. Так как реализуемый алгоритм имеет класс трудности NP, то 5 Гб глобальной памяти для вычисления функций роста групп большого порядка может быть недостаточно, поэтому в некоторых случаях, когда порядок группы велик, потребуется использовать внешнюю оперативную память, что отрицательно скажется на производительности программы. Это обусловлено тем, что взаимодействие GPU с оперативной памятью происходит посредством шины PCI Express 2.0 [6], которая имеет пиковую пропускную способность – 16 Гб/с, что во много раз меньше пиковой пропускной способности внутренней шины глобальной память акселератора (208 Гб/с). Очевидно, что полностью обозначенную проблему на уровне программной реализации решить не удастся, но есть возможность ее минимизировать за счет экономного использования глобальной памяти.

Во-вторых, стоит обратить внимание на возможность закрепления памяти [5]. Операционной системой (в данном случае Linux-ядром) используется концепция виртуальной памяти, следовательно, нельзя гарантировать, что обрабатываемые данные целиком расположены в оперативной памяти, так как

они могут быть разделены на страницы, часть которых в определенный момент времени может быть перемещена в долгосрочную память компьютера, например, на жесткий диск. Обращение к страницам виртуальной памяти, которые находятся в долгосрочной памяти компьютера требует дополнительных манипуляций для перемещения этих страниц в оперативную память, что, в свою очередь, требует дополнительного процессорного времени, кроме того, в такой ситуации шина PCI Express используется не полностью. Чтобы избежать подобных ситуаций существует возможность принудительно переместить все страницы виртуальной памяти, содержащие целевые данные, в оперативную память, то есть закрепить память.

В-третьих, необходимо отметить наличие поддержки универсального адресного пространства. Эта характеристика акселератора дает некоторые преимущества: сокрытие реализации механизмов работы с памятью и повторное использование программного кода. Оба преимущества обеспечиваются тем, что UVA [5] предоставляет единый программный интерфейс для работы как с переменными оперативной, так и с переменными глобальной памяти.

В-четвертых, важно, что максимальное количество запускаемых потоков не ограничено, это позволяет не обрабатывать и не предотвращать ситуации, которые могли бы возникнуть при выполнении программы, когда предел запускаемых потоков превышен.

4 Выбор аппаратно-программной платформы

На момент выполнения настоящей работы наиболее популярны следующие аппаратно-программные платформы, предназначенные для гибридных высокопроизводительных вычислений: CUDA [5], OpenCL [7], OpenACC [8].

При выборе какой-либо платформы решающее значение имеет именно аппаратное обеспечение. В данном случае имелась возможность использовать графический акселератор NVIDIA Tesla K20m, который поддерживается всеми перечисленными аппаратно-программными платформами.

4.1 OpenCL

OpenCL [7] – открытый стандарт для параллельного программирования, который предлагает эффективный способ использования возможностей гетерогенных многоядерных платформ.

Для представления программ OpenCL включает в себя специальный язык программирования – OpenCL C++, который является подмножеством языка C++ 14. Программы, написанные на языке OpenCL C++, являются кроссплатформенными, так как конструкции языка OpenCL C++ инкапсулируют подробности относительно низкоуровневой реализации механизмов взаимодействия с различными типами вычислительных устройств – это главное преимущество стандарта OpenCL, которое одновременно является и недостатком в контексте настоящей работы. К сожалению, унифицированные интерфейсы языка OpenCL C++ не предоставляют возможности создавать программы, которые ориентированы на архитектурные особенности той или иной аппаратной платформы, что препятствует раскрытию потенциала параллелизма какого-либо алгоритма в полной мере.

4.2 OpenACC

Стандарт OpenACC [8] предназначен для распараллеливания и портирования программ, написанных на языках Fortran, C и C++, на различные гетерогенные вычислительные системы. OpenACC-программы являются кроссплатформенными.

OpenACC не требует значительных затрат усилий на программирование, которые связаны именно с особенностями архитектуры конкретной вычислительной системы, так как подробности организации параллелизма для конкретных видов вычислительных систем сокрыты за специальными введенными стандартом OpenACC унифицированными директивами. Например, для программ, написанных на языках C и C++, указание OpenACC-директив происходит посредством директив препроцессора языка C. Благодаря концепции директив, OpenACC позволяет за короткие сроки преобразовать

последовательную программу в параллельную программу с незначительными модификациями.

Стандарт OpenACC имеет проблемы аналогичные проблемам OpenCL, но по сравнению с OpenCL, для OpenACC эти проблемы стоят особенно остро.

В силу высокого уровня абстракции директив OpenACC от архитектуры вычислительной системы, а также отсутствия специализированных для конкретной архитектуры инструментов и языковых конструкций, не представляется возможным эффективно вручную оптимизировать OpenACC-программы с учетом особенностей вычислительной системы, на которой предполагается выполнение программы, поэтому OpenACC-программы обычно уступают по производительности аналогичным CUDA-программам и OpenCL-программам. Этот факт является критическим аргументом против использования OpenACC для параллельной реализации модифицированного алгоритма A-I, поэтому другие особенности OpenACC в рамках настоящей работы рассматривать не имеет смысла.

4.3 CUDA

CUDA [5] – программно-аппартная платформа, предназначенная для организации GPGPU на гибридных вычислительных системах с графическими акселераторами компании NVIDIA.

CUDA предназначена для работы только с видеокартами компании NVIDIA и предоставляет нативную поддержку и широкий специализированный именно для графических акселераторов компании NVIDIA набор технологий и инструментов, в который входят: nvcc – компилятор языка CUDA C, nvprof – профилировщик времени CUDA-программ, cuda-memcheck – профилировщик пространства CUDA-программам [5].

Для представления программ CUDA включает в себя специальный язык программирования – CUDA C/C++ (или CUDA C), который является расширением языков C и C++. Язык CUDA C имеет относительно простую грамматику, не смотря на это, считается, что программы на этом языке более выразительны, чем программы на языке платформы OpenCL. Также считается, что обычно CUDA-программы превосходят по производительности аналогичные OpenCL-программы.

Кроме того, необходимо отметить косвенные, но не менее важные, положительные черты CUDA: наличие качественной документации, а также актуальной профильной литературы.

Не смотря на все преимущества, CUDA имеет ряд недостатков: не представляется возможным переработать исходный код последовательной программы без значительных изменений; не представляется возможным переносить CUDA-программы на другие аппаратные платформы без акселераторов NVIDIA. Можно считать, что второй из указанных недостатков CUDA практически нивелирован за счет того, что оборудование компании

NVIDIA пользуется большой популярностью в сфере высокопроизводительных вычислений [9].

Для решения задач, поставленных в рамках настоящей работы, указанные недостатки платформы CUDA не являются критическими на фоне всех ее преимуществ, поэтому выбор CUDA для реализации модифицированного алгоритма A-I является приемлемым.

5 Программная реализация алгоритма

5.1 Декомпозиция алгоритма

В целях снижения сложности структуры программной реализации модифицированного алгоритма A-I необходимо выделить его основные операции, которые повторяются от итерации к итерации, а также для удобства дать этим операциям имена: вычисление очередного элемента $g = p \cdot x$ группы G (листинг 2, строки 6-7) – MULTIPLY, добавление g в шар K_s и маркировка g посредством вектора V (листинг 2, строки 8-9) – MARK, вычисление мощности разности $P = K_s – K_{s–1}$ (листинг 2, строки 10-11) – DELETE.

Удобно представить модифицированный алгоритм A-I в виде укрупненной блок-схемы, на которой основные операции изображены как блоки-подпрограммы (рисунки 1-4).

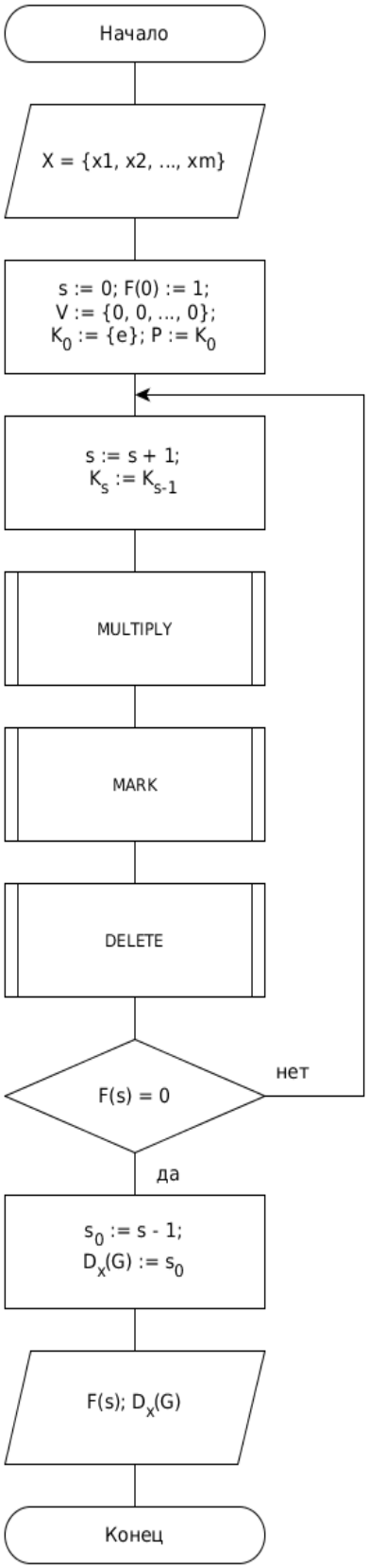

Рисунок 1 – Блок-схема
модифицированного алгоритма A-I

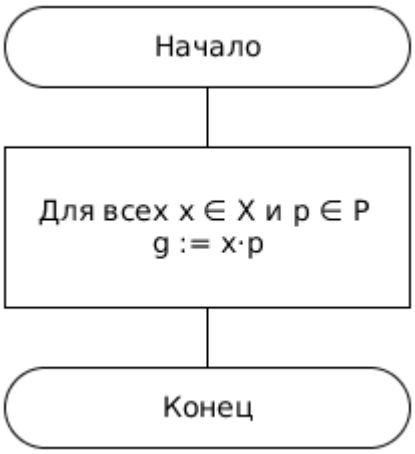

Рисунок 2 – Блок-схема
операции MULTIPLE

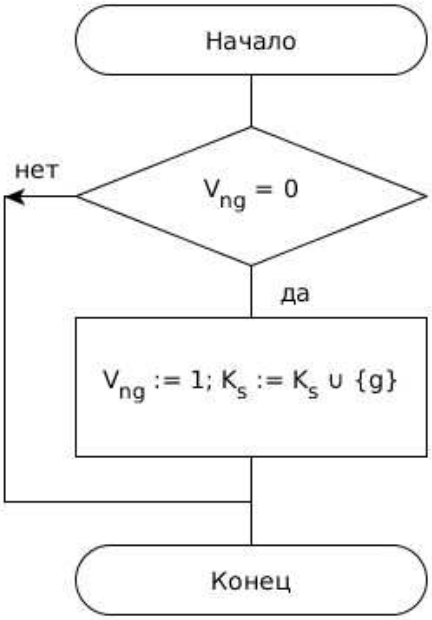

Рисунок 3 – Блок-схема операции
MARK

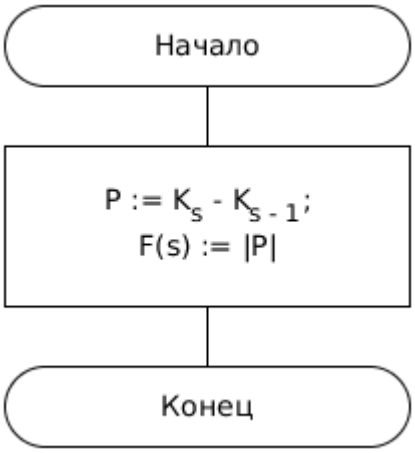

Рисунок 4 – Блок-схема
операции DELETE

5.2 Программная интерпретация математических объектов

При написании программного кода на языке CUDA C возможно использовать стандартную библиотеку языка C++ [10, с. 295-296], которая содержит в себе пригодные для реализации модифицированного алгоритма A-I структуры данных, соответствующие математическим объектам (например, множества и векторы), а также алгоритмы для их обработки, которые реализуют специальные математические операции, но в контексте требований, поставленных в рамках настоящей работы использование этих структур данных неприемлемо по ряду причин: такие структуры данных требуют избыточных вычислительных ресурсов (памяти и процессорного времени), по сравнению с примитивными структурами данных; имеют относительно сложные внутреннюю структуру и механизмы работы, поэтому не очевидно каким образом эти структуры данных взаимодействуют с аппаратным обеспечением, что препятствует созданию программы, эффективно утилизирующей ресурсы конкретной аппаратной платформы. Недостатки указанных структур данных обусловлены их относительно высоким уровнем абстракции, которая, в свою очередь, является следствием использования концепций объектно-ориентированного программирования при их реализации.

Основными объектами алгоритма (листинг 2) являются шары K_s и K_{s-1}, множество P, а также вектор V. Эти объекты тривиально и эффективно (по объему памяти, что в данном случае важно) реализуемы в программном коде с помощью структуры данных – массив [11, с. 85]. Такие объекты как элементы, принадлежащие K_s, K_{s-1}, V и P, в таком случае будут являться обыкновенными переменными.

Пусть есть два массива – K и V, соответствующие указанным математическим объектам (шарам K_s и K_{s-1} и вектору V), тогда отношение

между элементами массивов K и V можно интерпретировать как ассоциативный массив K-V с ключами-дубликатами, в котором элементы массива K – ключи, а элементы массива V – значения. В таком случае каждому элементу g массива K соответствует элемент массива V с индексом g.

Таким образом, при использовании ассоциативного массива K-V фактически не требуется сохранять неуникальные элементы массива K между итерациями, так как все вычисленные на предыдущих итерациях алгоритма элементы массива K будут помечены посредством массива V по указанному ранее принципу (см. листинг 2), что в перспективе значительно сокращает вычислительную пространственную сложность программной реализации алгоритма, при условии, если объем оперативной памяти, требуемый для хранения одного элемента массива V меньше аналогичного объема для хранения одного элемента массива K. Кроме того, при таком подходе нет необходимости введения самостоятельного массива P, предназначенного для хранения уникальных элементов из массива K.

Описание модифицированного алгоритма A-I, представленное посредством листинга 2, также имеет ряд специальных математических операций, среди которых коммутативная композиция (или произведение) элементов множеств посредством полиномов Холла (строка 7, $g := p \cdot x$), объединение шара и множества единичной мощности (строка 9, $K_s := K_s \cup \{g\}$), вычисление разности шаров (строка 10, $P := K_s - K_{s-1}$), а также вычисление мощности множества (строка 11, $F(s) := |P|$). Все эти операции не имеют нативной реализации на уровне языковых конструкций CUDA C, например, в отличие от арифметических операций («+», «-», «*», «/»). Кроме того, указанные операции имеют декларативный характер, что вызывает некоторую неоднозначность при их программной реализации с помощью императивного языка программирования CUDA C.

С учетом выбранных для реализации объектов K_s, K_{s-1}, V и P структур данных указанные операции эффективно реализуются в программном коде следующим образом: композиция элементов множеств ($g := p \cdot x$) как совокупность операций целочисленной компьютерной математики; объединение шара и множества единичной мощности ($K_s := K_s \cup \{g\}$) как запись значения на свободное место в массиве K; вычисление разности шаров ($P := K_s - K_{s-1}$) как удаление (перемещение в левую часть массива) неуникальных элементов из массива K; вычисление мощности множества ($F(s) := |P|$) как подсчет с помощью переменной-счетчика элементов массива K (при удалении неуникальных элементов).

В соответствии с принятой интерпретацией математических объектов модифицированного алгоритма A-I построены блок-схемы (рисунки 5-8), которые более подробно раскрывают некоторые тезисы, использованные в предыдущих абзацах. Пояснение к рисункам 5-8: F – массив для хранения значений функции роста; p – переменная-счетчик для подсчета уникальных

элементов массива K (не путать с элементом множества p ∈ P); len(F) – длина массива F; s – переменная-счетчик цикла.

Необходимо обратить внимание на то, что на рисунке 6 бинарная операция композиции изменяет как левый, так и правый свои операнды.

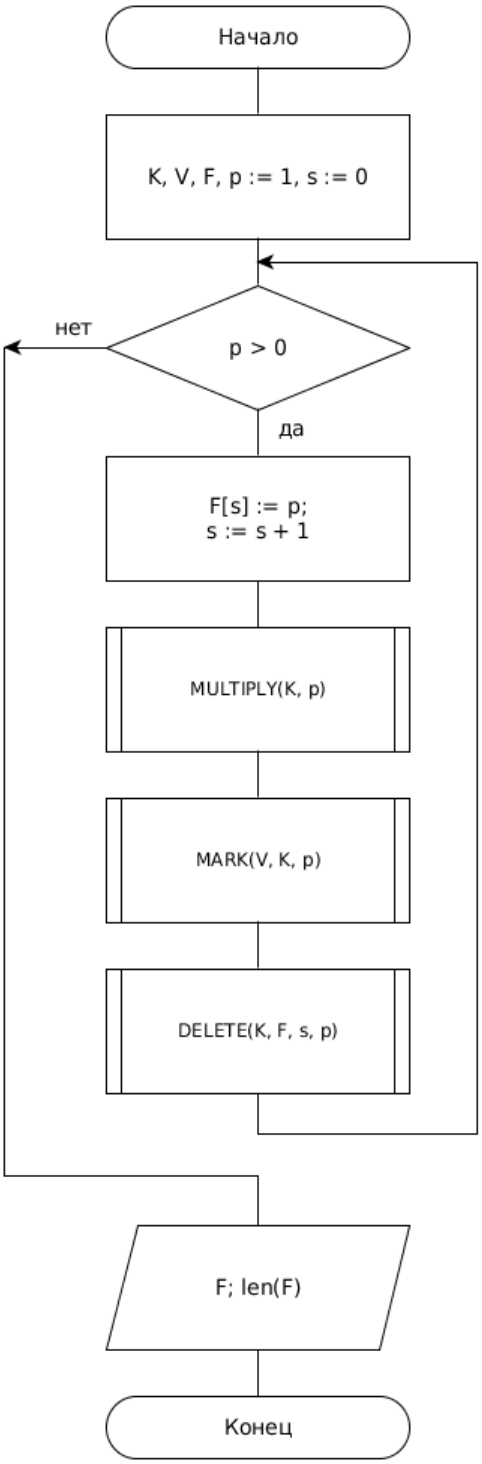

Рисунок 5 – Блок-схема
модифицированного алгоритма A-I

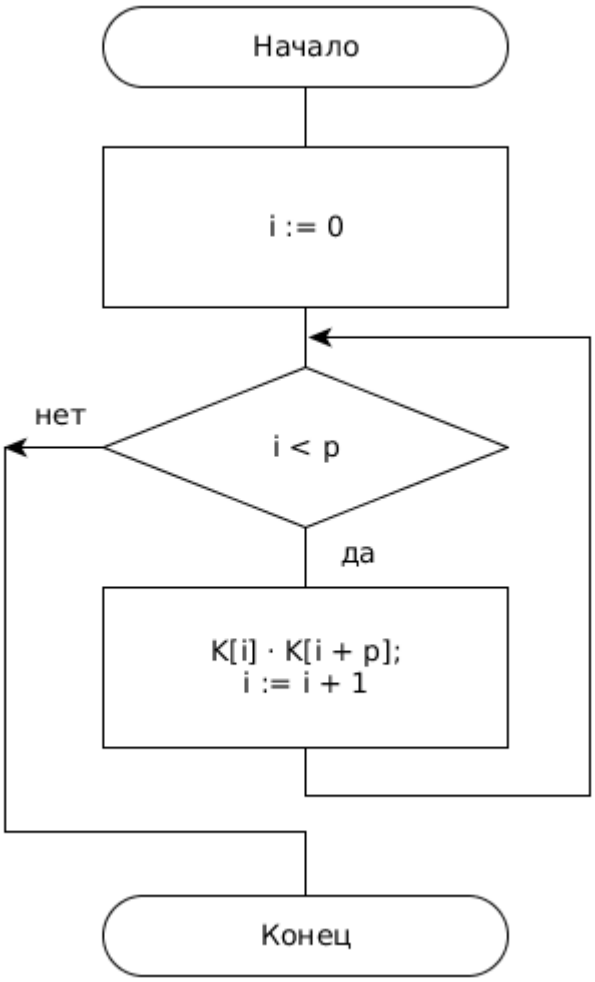

Рисунок 6 – Блок-схема операции MULTIPLY

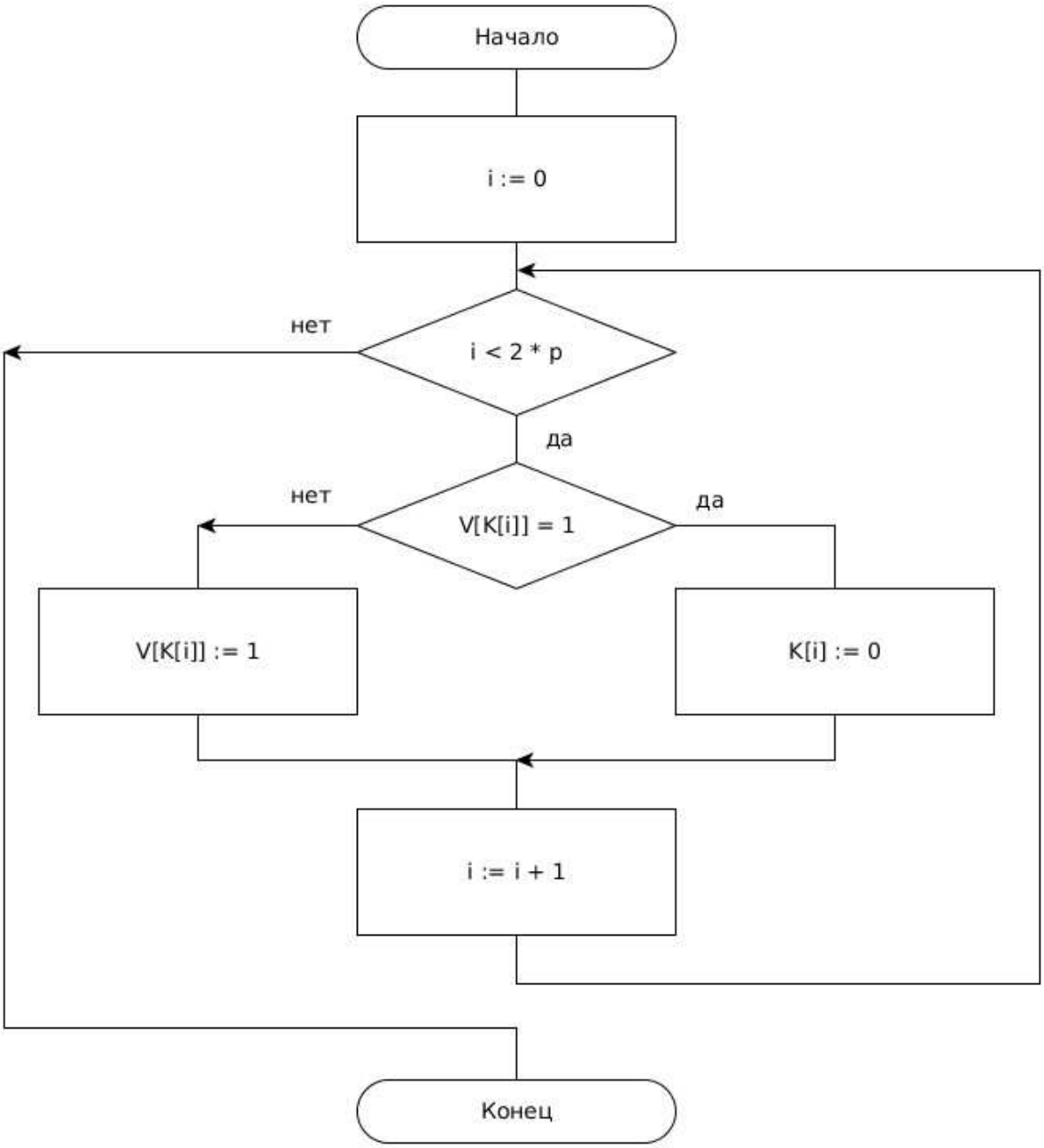

Рисунок 7 – Блок-схема операции MARK

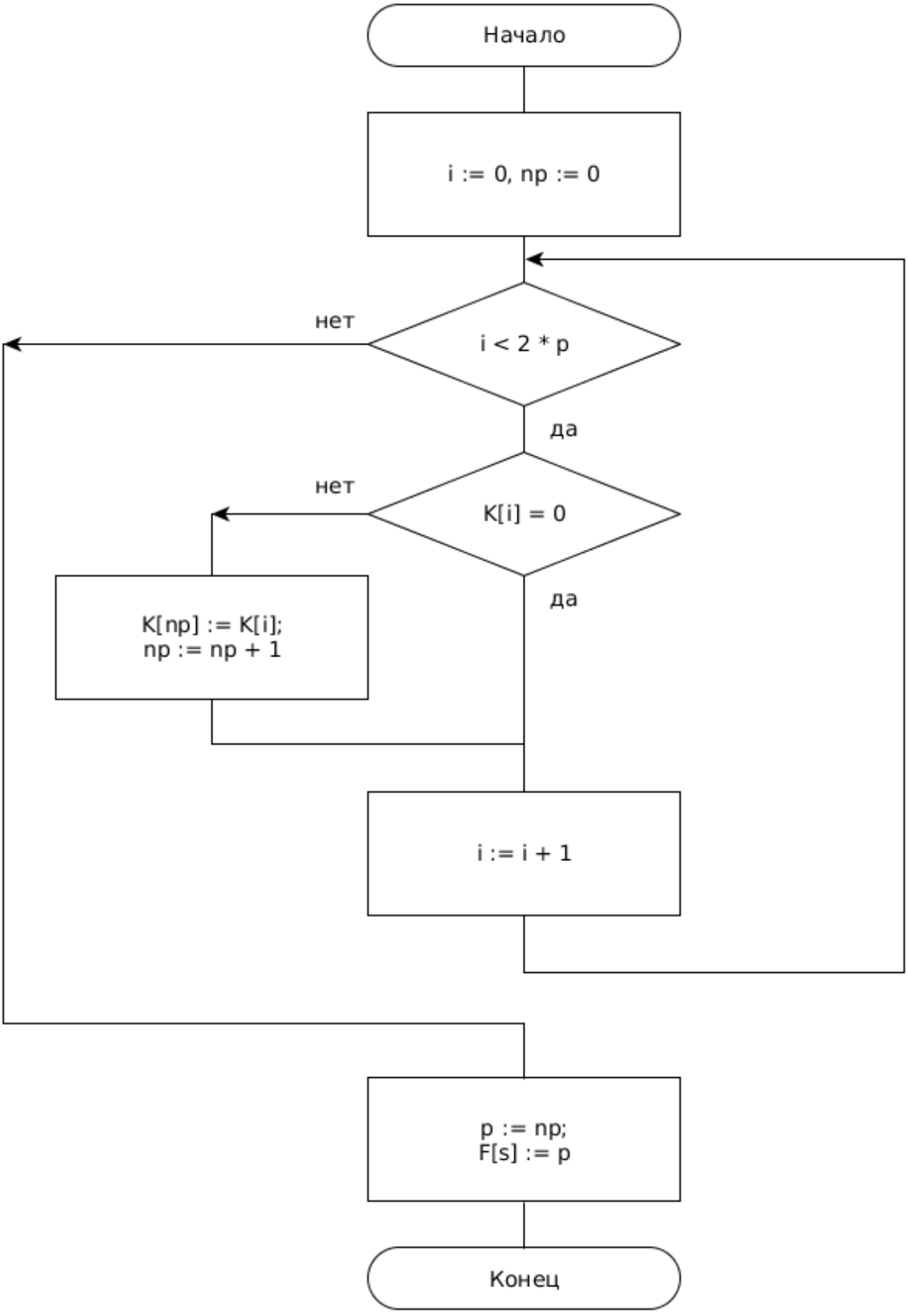

Рисунок 8 – Блок-схема операции DELETE

5.3 Поиск перспективных участков для распараллеливания

Далее рассматриваются перспективные для распараллеливания по данным (согласно SIMD [12, с. 125]) участки модифицированного алгоритма A-I без учета особенностей платформы CUDA. Все аспекты, касающиеся особенностей CUDA, рассматриваются в разделе – «CUDA-реализация алгоритма». Такое разделение необходимо для соблюдения системного подхода в изучении модифицированного алгоритма A-I.

Модифицированный алгоритм A-I имеет высокую пространственную сложность и, следовательно, объем памяти, требуемой реализацией этого алгоритма критичен, поэтому все варианты распараллеливания данного алгоритма, которые требуют дополнительных затрат памяти были отклонены и далее не приводятся.

Для оценки потенциала SIMD-параллелизма того или иного участка алгоритма удобно использовать какой-либо численный показатель, поэтому далее используется метрика SIMD-параллелизма Pd, которая показывает, какую часть (в процентах) некоторого массива данных можно обработать параллельно.

Наиболее перспективными для распараллеливания по данным являются следующие операции модифицированного алгоритма A-I: MULTIPLY – может быть распараллелена без каких либо ограничений по паттерну «map» [12, с. 128-130], то есть Pd(MULTIPLY) = 100%; MARK – может быть распараллелена с ограничениями по шаблону «map», так как не исключена возможность коллизий при извлечении значений из ассоциативного массива K-V в силу того, что до начала операции DELETE не гарантируется, что все элементы K уникальны, то есть в K могут содержатся одинаковые значения (см. рисунки 9-10).

Чтобы полностью исключить коллизии при извлечении значений из ассоциативного массива K-V возможно использовать атомарные операции (рисунок 11, где атомарный доступ обозначен пунктирной линией) или поступить описанным далее образом. Так как известно, что внутри каждого участка массива K по p элементов не может быть одинаковых значений, то существует возможность распараллелить MARK для каждого такого участка по отдельности. В этом случае потребуется одна SIMD-инструкциями [12, с. 127-128] на каждую половину массива K (рисунок 12).

К сожалению, нельзя наверняка сказать, какой из вариантов распараллеливания MARK более эффективен, так как эффективность того или иного варианта полностью определяется соответствующими конкретными аппаратно-программными реализациями. Не смотря на это, если опустить подробности реализации атомарных операций, то вариант с атомарными операциями более предпочтителен.

Можно определить теоретический худший случай для первого варианта (с атомарными операциями), когда происходит максимальное количество коллизий при доступе к значениям ассоциативного массива K-V: пусть каждому элементу

массива V соответствуют два элемента-ключа из массива K (по одному из каждой части размером p) с индексами меньшими, чем 2 · p, тогда если каждый поток обрабатывает каждый такой элемент массива K, то происходит по два последовательных обращения к каждому элементу массива V. В этом худшем случае Pd(MARK) = 50%. Очевидно, что во всех остальных случаях (когда коллизий происходит меньше) Pd(MARK) > 50%.

Для второго варианта (с последовательной обработкой половин массива K) Pd(MARK) = 50% всегда, так как две части массива K размером p в любом случае обрабатывается последовательно.

К сожалению, потенциал к распараллеливанию по данным без использование дополнительных переменных операции DELETE стремится к нулю, так как эта операция представляет собой перемещение элементов внутри одного массива, которое не имеет тривиальной параллельной реализации. Исходя из этого факта, принято решение – ввести две дополнительные переменные для каждого потока. Первая переменная предназначена для сохранения новой позиции (индекса) ненулевого элемента массива K, а вторая – для сохранения значения этого элемента. Таким образом, механизм работы каждого потока в операции DELETE следующий: если элемент, проверяемый потоком, ненулевой, то инкрементировать p и сохранить предыдущее значение p в качестве новой позиции элемента (эти операции происходят атомарно, нумерация позиций начинается с нуля); сохранить значение элемента; синхронизироваться; записать значение сохраненного элемента в массив K согласно новому индексу.

В силу того, что в операции DELETE требуется выполнять барьерную синхронизацию, уровень SIMD-параллелизма в два раза меньше максимального возможного, то есть Pd(DELETE) = 50%.

Схематическое описание механизма работы потока в операции DELETE алгоритма показано на рисунке 13.

Значения метрики SIMD-параллелизма каждой из операций модифицированного алгоритма A-I можно увидеть в таблице 2.

Таблица 2 – Оценки SIMD-параллелизма операций модифицированного алгоритма A-I

Операция	Pd в лучшем случае, %	Pd в худшем случае, %
MULTIPLY	100	100
MARK (вариант 1)	100	50
MARK (вариант 2)	50	50
DELETE	50	50

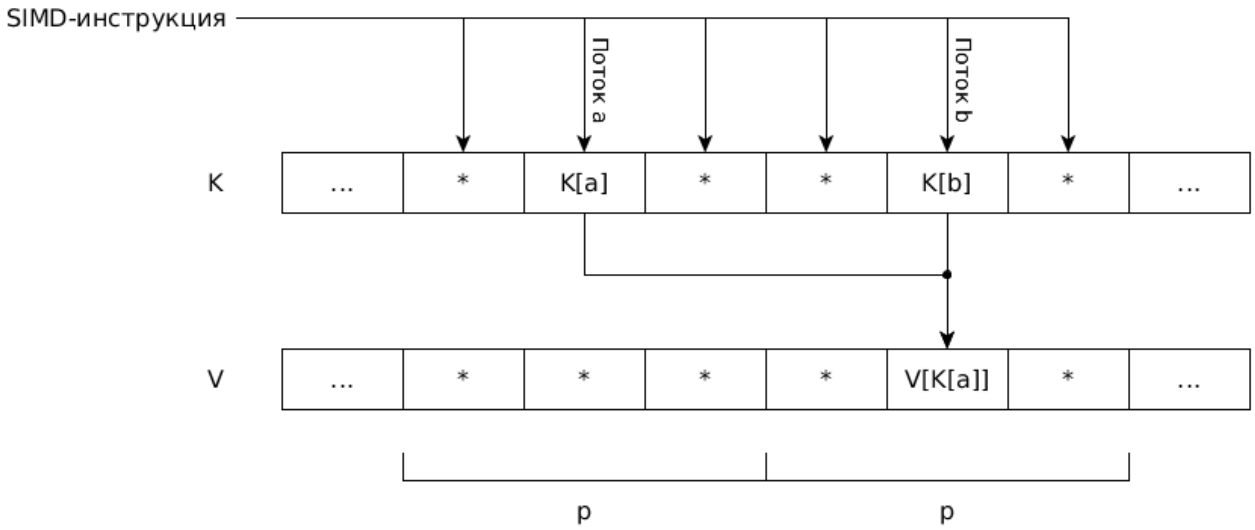

Рисунок 9 – Коллизия в массиве K-V (эффект гонок потоков)

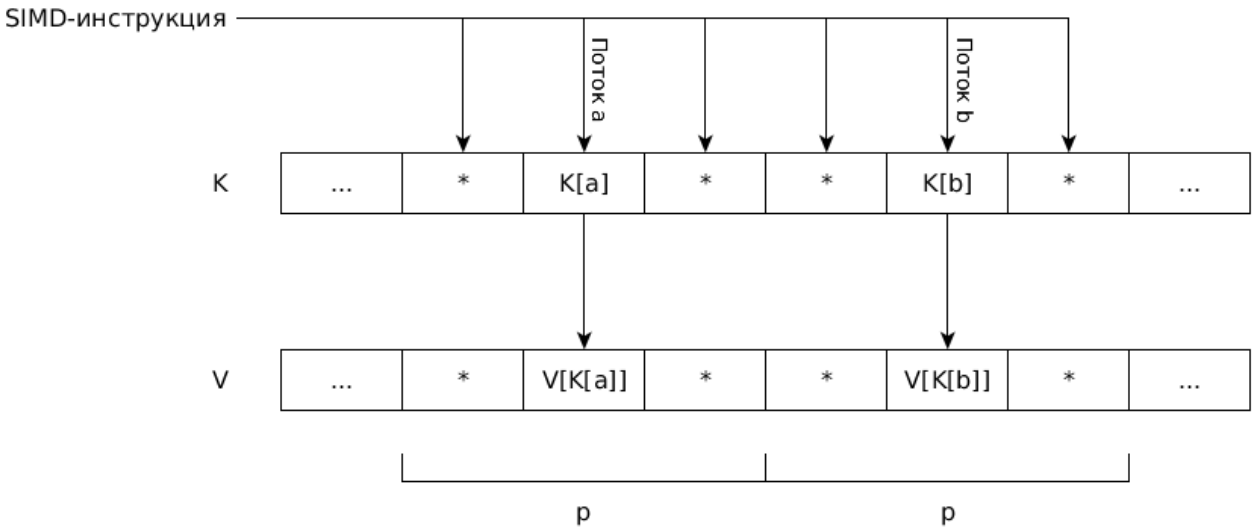

Рисунок 10 – Обработка массива K-V без коллизий

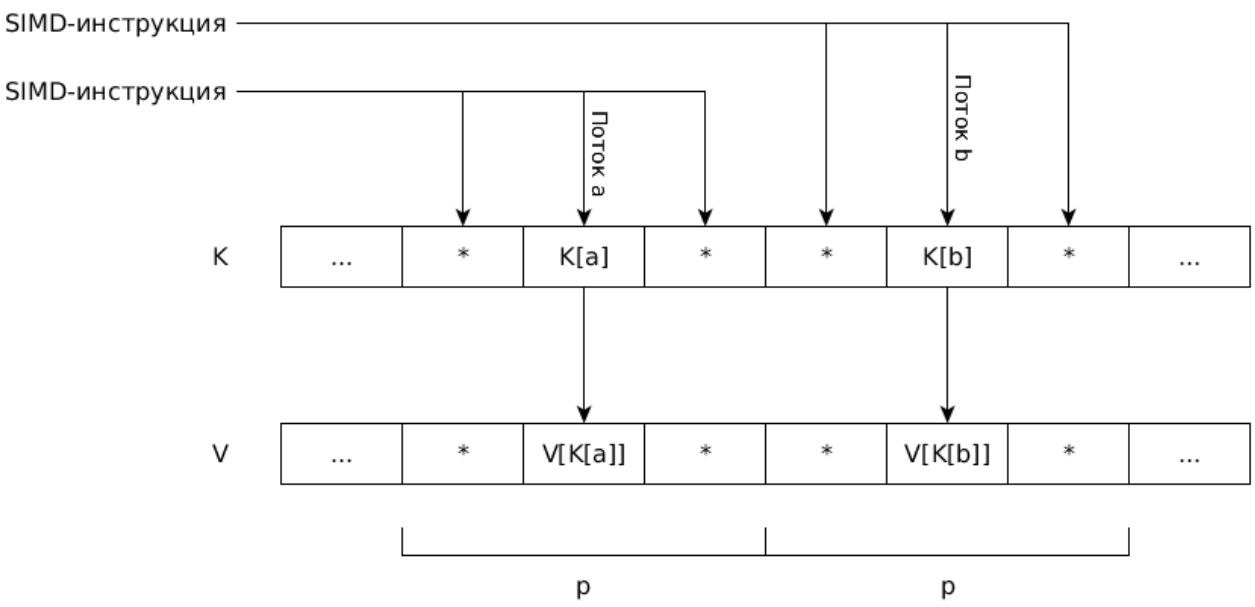

Рисунок 11 – Обработка массива K-V за 2 SIMD-инструкции

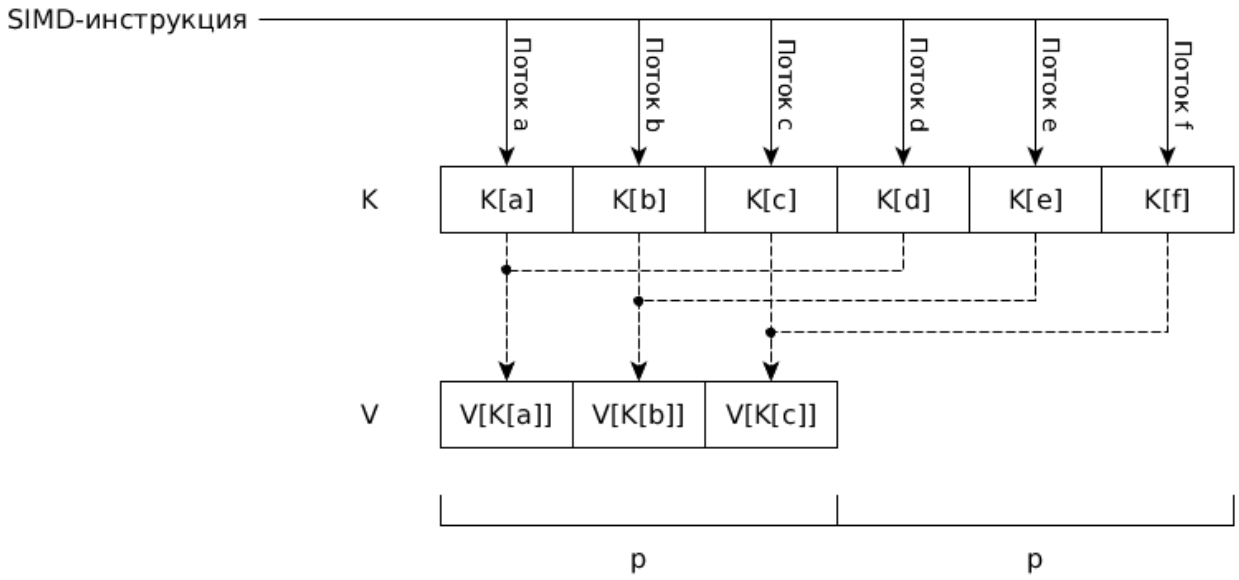

Рисунок 12 – Обработка массива K-V с помощью атомарной операции

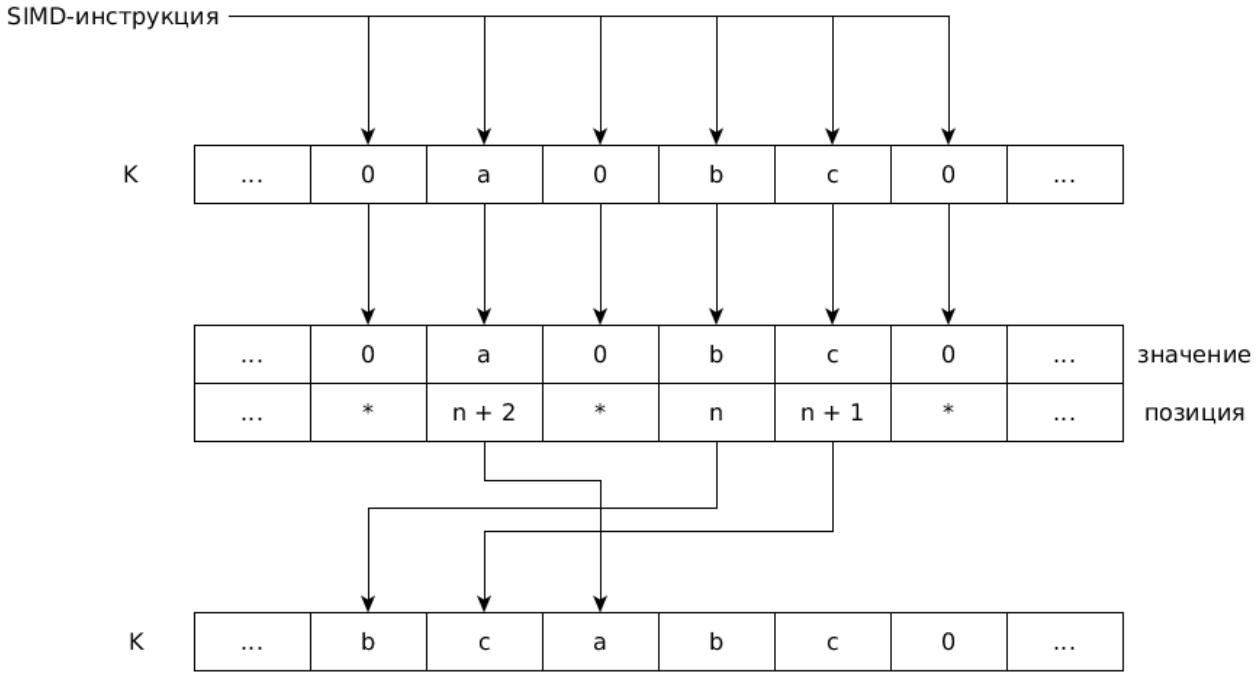

Рисунок 13 – Параллельная перестановка элементов массива K

5.4 Парадигма программирования

Исходные тексты программ на языке CUDA C могут содержать фрагменты кода на языке C++, поэтому можно сказать, что для реализации модифицированного алгоритма A-I возможно придерживаться концепций как процедурной и структурной парадигм программирования, так и объектно-ориентированного программирования.

Для программной реализации модифицированного алгоритма A-I была выбрана структурная парадигма программирования в силу ряда причин: в полной мере поддерживается языком CUDA C; обладает относительно низким (по сравнению с объектно-ориентированным программированием) и в то же время достаточным (по сравнению с процедурным программированием) уровнем абстракции, поэтому не требует накладных расходов вычислительных ресурсов, а также позволяет более эффективно взаимодействовать с аппаратным обеспечением, что позволяет получить программу, обладающую высоким «коэффициентом полезного действия», хотя имеет и незначительный недостаток – ограничивает применение эволюционной концепции [13] разработки программного обеспечения (например, по сравнению с ООП), не смотря на это, возможности, предоставляемые структурной парадигмой программирования являются достаточными для достижения целей, поставленных в рамках настоящей работы.

5.5 CUDA-реализация алгоритма

Все вычисления CUDA-программ обычно сосредоточены в CUDA-ядрах [5], что справедливо и для CUDA-реализации модифицированного алгоритма A-I, поэтому далее подробно описаны аспекты программирования CUDA-ядер, каждое из которых реализует одну из операций модифицированного алгоритма A-I. Аспекты, касающиеся межъядерного взаимодействия достаточно подробно описаны с помощью блок-схемы, изображенной на рисунке 5.

5.5.1 Типы данных

Предполагается, что в массив K могут быть записаны целые неотрицательные числа большого порядка (порядок более 10 в десятичной системе счисления), поэтому тип элементов массива K должен быть целочисленным беззнаковым максимального размера. В языке CUDA C этим требованиям отвечает 8-байтовый тип unsigned long long [5], который позволяет сохранять числа до 20 порядка (в десятичной системе счисления).

Тип значений массива V должен иметь минимальный возможный размер, который пригоден для представления двух состояний, то есть один элемент массива V должен быть 1-битным: бит установлен – элемент массива K помечен; бит не установлен – элемент массива K не помечен. К сожалению, в языке CUDA C нет 1-битного типа данных, а также биты одной переменной не могут быть обработаны параллельно (независимо разными потоками). Один из типов наименьшего размера, который возможно использовать при написании исходного кода программы на языке CUDA C – unsigned char [5] (1-байт). Очевидно, что размер типа unsigned char избыточен для представления одного значения ассоциативного массива K-V, но лишь использование этого типа позволяет организовать обработку массива K-V с максимальным уровнем параллелизма (первый подход); альтернативой является использование каждого бита переменной массива V в качестве значения ассоциативного массива K-V, но при таком подходе требуется обеспечить атомарный доступ к одной переменной массива V, что, в свою очередь, снижает уровень параллелизма (Pd) в худшем случае в число раз, равное битности типа данных одного элемента массива V (второй подход). Так как атомарные операции [5; 14, с. 42] языка CUDA C могут быть применены к 32-битным и 64-битным словам, то при последнем подходе размер одного элемента массива V должен быть равен 4 байтам, что соответствует, например, элементу типа unsigned long, и, следовательно уровень параллелизма обработки ассоциативного массива K-V в 32 раза меньше аналогичного уровня для первого подхода. Не смотря на относительно низкий уровень параллелизма, второй подход имеет ряд существенных преимуществ: во-первых, объем глобальной памяти, требуемый для размещения массива V оптимален и в 8 раз меньше аналогичного объема памяти, требуемого вариантом без использования атомарной операции; во-вторых, трафик через шину глобальной памяти, при обращении к массиву V в 8 раз меньше аналогичного трафика, требуемого вариантом без использования

атомарной операции. Очевидно, что второй подход наиболее эффективен в отношении памяти, но нельзя однозначно утверждать, что он приемлем в отношении времени выполнения, поэтому необходимо реализовать оба подхода и выбрать в целом наиболее оптимальный из них на основе результатов вычислительных экспериментов.

5.5.2 Разработка CUDA-ядер

Модель вычислений CUDA реализует концепцию SIMT [14, с. 19], которая унаследовала основные идеи SIMD, не смотря на это, имеются принципиальные различия между этими концепциями параллельных вычислений. Согласно SIMT все нити разбиты на группы по 32 нити, называемые варпами. Концепцией SIMT гарантируется, что только нити одного варпа выполняются физически одновременно, при этом нити разных варпов могут находится на разных стадиях выполнения. Помимо группировки в варпы нити также группируются в одномерные, двумерные или трехмерные блоки одинакового размера и одинаковой размерности, которые, в свою очередь, образуют сетку блоков соответствующей размерности. В рамках модели SIMT одним из ключевых понятий является ядро – абстракция SIMD-инструкции, которая реализуется в виде специальной функции языка CUDA C.

Потенциал параллелизма операции MULTIPLY с помощью CUDA-ядра реализуется аналогично SIMD-решению (листинг 3). Для реализации операции коммуникативного умножения элементов групп с помощью полиномов Холла, которая требуется в MULTIPLE, использована кодовая база из источника [15] (язык программирования C++), переписанная на язык CUDA C.

Реализация параллелизма операций MARK и DELETE в рамках модели SIMT имеет ряд важных аспектов.

Как и в модели SIMD операцию MARK в модели SIMT можно реализовать либо с использованием атомарной операции, либо с помощью условного разделения массива K на две половины, которые будут обрабатываться последовательно относительно друг друга. С большой вероятностью первый вариант (с атомарной операцией) окажется более эффективным в рамках модели SIMT, так как атомарные операции в CUDA выполняются достаточно быстро (за счет аппаратного планирования), например, по сравнению с OpenMP, но без непосредственного сравнения времени выполнения ядер, организованных в соответствии с обоими вариантами, этого гарантировать нельзя, поэтому необходимо создать две версии ядра операции MARK и выбрать наиболее эффективное из них (листинги 4-5).

Листинг 3 – Исходный код ядра операции MULTIPLY

```
1        __global__ void
2         __launch_bounds__(MAX_THREADS_PER_BLOCK)
3         kernel_mul_greedy(uint64 *K, uint64 p, uint64 work_size) {
```

```
4
5           uint32 tid = GET_1D_TID();
6
7           if (tid < work_size) {
8               __shared__ uint64 k[2][MAX_THREADS_PER_BLOCK];
9
10              k[0][threadIdx.x] = K[tid];
11              k[1][threadIdx.x] = K[tid + p];
12
13              b0_product_2gens(k[0] + threadIdx.x, k[1] + threadIdx.x);
14
15              K[tid] = k[0][threadIdx.x];
16              K[tid + P] = k[1][threadIdx.x];
17          }
18      }
```

Листинг 4 – Исходный код ядра операции MARK (версия 1)

```
1       __global__ void
2        __launch_bounds__(MAX_THREADS_PER_BLOCK)
3        kernel_mark_greedy(uint64 *K, uint32 *V, uint64 work_size) {
4
5           uint64 tid = GET_1D_TID();
6
7           if (tid < work_size) {
8               uint64 k = K[tid];
9               uint64 k_div_32 = k >> 5;
10              uint32 mask_32 = 0x80000000 >> (k & 0x1F);
11
12              if (atomicOr(V + k_div_32, mask_32) & mask_32) {
13                  K[tid] = 0;
14              }
15          }
16      }
```

Листинг 5 – Исходный код ядра операции MARK (версия 2)

```
1       __global__ void
2        __launch_bounds__(MAX_THREADS_PER_BLOCK)
3        kernel_mark(uint64 *K, uchar8 *V, uint64 work_size) {
4           uint64 tid = GET_1D_TID();
5
6           if (tid < work_size) {
7               uint64 k = K[tid];
8
9               if (V[k]) {
10                  K[tid] = 0;
11              }
12              else {
13                  V[k] = 1;
14              }
15          }
```

```
16      }
```

Для удобства в исходном коде программы вместо «unsigned long long» используется алиас «uint64», вместо «unsigned int» – алиас «uint32», вместо «unsigned char» – «uchar8».

После серии проведенных вычислительных экспериментов стало очевидно, что первый вариант (с атомарным доступом) реализации операции MARK более эффективен как по объему памяти, так и по времени выполнения: время выполнения первой версии ядра операции MARK приблизительно в 1,5 раза меньше времени выполнения второй версии ядра этой операции.

Как отмечалось ранее, реализация ядра операции DELETE требует барьерной синхронизации всех потоков. Так как синхронизация потоков на уровне нескольких блоков в CUDA нативно не поддерживается (синхронизация нитей [5] возможна только в пределах одного блока), то размер сетки при запуске соответствующего ядра ограничен одним блоком. Возможно осуществлять запуск ядра последовательно несколько раз с сеткой размером 1 блок, что, в свою очередь, приводит к дополнительной принудительной синхронизации при завершении выполнения ядра, а также ко множеству последовательных вызовов функций CUDA API, которые отвечают за подготовку к запуску и запуск ядра. Чтобы избежать многократных последовательных вызовов функций CUDA API и принудительной синхронизации, совокупное время выполнения которых может быть достаточно большим, более рационально в данном случае использовать другой подход – паттерн «перманентный поток» (англ. «persistent thread» [16]). Согласно указанному паттерну ответственность за распределение работы берет на себя само ядро: нити, которые обработали одну часть данных, не завершаются, а продолжают обрабатывать другую (необработанную) часть данных; благодаря этому потребность в последовательном вызове ядра с принудительной синхронизацией и, следовательно, функций CUDA API отпадает. С учетом особенностей SIMT необходимо скорректировать оценку SIMD-параллелизма операции DELETE: $Pd(DELETE) = (100\% \cdot 2 \cdot p) / (0{,}5 \cdot N)$, N – размер блока потоков. Очевидно, что при запуске ядра операции DELETE следует выбрать максимальный возможный размер блока (1024 потока на блок для Tesla K20m), поэтому оценку SIMD-параллелизма операции DELETE для акселератора Tesla K20m можно записать следующим образом: $Pd(DELETE) = (100\% \cdot 2 \cdot p) / 512$.

Исходный код функции-ядра операции DELETE представлен посредством листинга 6.

Листинг 6 – Исходный код ядра операции DELETE

```
1       __global__ void
2         __launch_bounds__(MAX_THREADS_PER_BLOCK)
3         kernel_del_persistent(uint64 *K, uint64 *p, uint64 work_size) {
4
5           uint64 tid = GET_1D_TID();
```

```
6       uint64 task_id;
7       uint64 pos;
8       uint64 k;
9
10      #if UNROLL_LOOPS
11        #pragma unroll 512
12      #endif
13      for (uint32 i = 0; i < work_size / blockDim.x + 1; i++) {
14        task_id = tid + i * blockDim.x;
15        if (task_id < work_size) {
16          k = K[task_id];
17
18          if (k) {
19            pos = atomicAdd(p, 1);
20          }
21
22          __syncthreads();
23
24          if (k) {
25            K[pos] = k;
26          }
27        }
28        else {
29          return;
30        }
31      }
32    }
```

5.5.3 Оптимизация CUDA-ядер

Выделяют следующие стратегии оптимизации CUDA-программ: 1 – оптимизация доступа к памяти, оптимизация математики, оптимизация параллелизма. Каждая из стратегий предназначена для решения специфических для платформы CUDA проблем. Классификация наиболее распространенных проблем [5; 14, с. 152], которые характерны для CUDA-программ, представлена посредством таблицы 3.

Таблица 3 – Классификация проблем CUDA-программ [5; 14, с. 152]

Проблемы доступа к памяти	Проблемы параллелизма	Проблемы математики
Спиллинг регистров	Дефицит ресурсов для выполнения потоков	Операции с вещественными числами одинарной точности
Ограничение коалесинга	Дивергенция нитей варпа	Операции с вещественными числами двойной точности
Избыточные операции с оперативной памятью	–	–
Избыточные операции с глобальной памятью	–	–

Возможность присутствия у CUDA-реализации модифицированного алгоритма A-I проблем, связанных с вещественной арифметикой, можно исключить, так как модифицированный алгоритм A-I не требует выполнения каких-либо операций с вещественными числами. Также можно исключить возможность присутствия избыточных операций с оперативной памятью, так как во время выполнения программы в оперативною память компьютера копируется лишь одна переменная (по указателю d_P) из глобальной памяти видеокарты. Кроме того, можно исключить, проблему избыточных обращений в глобальную память, так как ее влияние было минимизировано при проектировании структуры ядер.

Другие проблемы, перечисленные в таблице 3, имеют место быть.

5.5.3.1 Спиллинг регистров

Спиллинг (или разлив) [5] регистров CUDA-ядра происходит, когда при компиляции этого ядра достигнут предел доступных для одного потока регистров (64 регистра для архитектуры Kepler); в этом случае компилятор помечает переменные, которые не удалось поместить в регистровый файл, атрибутом принадлежности к локальной памяти, которая обладает большей латентностью доступа, сравнимой с латентностью доступа к глобальной памяти. В следствие описанного явления общая производительность ядра снижается.

Для выявление спиллинга программа была скомпилирована в специальном режиме компилятора nvcc (аргумент -Xptxas="-v"), при котором выводится отладочная информация, в том числе и информация о количестве используемых каждым ядром регистров. Установлено, что ни одно ядро не требует более 20 регистров на поток, что меньше максимального допустимого количества регистров на поток для акселератора Tesla K20m (64 регистра), следовательно, спиллинга регистров ни одного из ядер не наблюдается.

5.5.3.2 Ограничение коалесинга

Все транзакции глобальной памяти [5] акселератора происходят независимо для каждой половины варпа (по 16 потоков). Транзакции полу-варпа объединяются в один запрос, если выполняется ряд условий [5; 14, с. 52]: во-первых, все потоки обращаются к последовательно расположенным в памяти словам размером 1, 4, 8 или 16 байтов; во-вторых, первый поток полу-варпа обращается по адресу выровненному по размеру образованного этими словами блока данных, то есть для слов размером 1, 4, 8 и 16 байтов адреса блоков должны быть кратны 16, 64, 128 и 256 соответственно. Такое объединение транзакций принято называть коалесингом (слиянием). Коалесинг позволяет сократить число обращений к глобальной памяти графического акселератора и увеличить реальную пропускную способность шины глобальной памяти.

К сожалению, все ядра CUDA-реализации модифицированного алгоритма A-I соответствуют условиям выполнения коалесинга лишь частично, поэтому транзакции глобальной памяти во время выполнения программы объединяются не в один запрос, а в несколько запросов. Этот факт подтверждается результатами профилирования программы с помощью nvprof (таблицы 4-5).

Таблица 4 – Статистика чтения глобальной памяти

Ядро	Минимальное число транзакций в одном запросе	Максимальное число транзакций в одном запросе	Среднее число транзакций в одном запросе
kernel_mul_greedy	1	2,5	2,16
kernel_mark_greedy	1	2	1,85
kernel_del_persistent	1	2	1,85

Таблица 5 – Статистика записи в глобальную память

Ядро	Минимальное число транзакций в одном запросе	Максимальное число транзакций в одном запросе	Среднее число транзакций в одном запросе
kernel_mul_greedy	1	2,5	2,16
kernel_mark_greedy	0	2	1,85
kernel_del_persistent	0	19	12,17

Ограничение коалесинга для ядер операций MULTIPLY и MARK обусловлено особенностями модифицированного алгоритма A-I: во-первых, алгоритм требует выполнения обращений к элементам массива K, которые расположенны не последовательно, (MULTIPLY; листинг 3, строки 10-11 и 15-16); во-вторых, алгоритм требует выполнения перекрестных обращений в массив V (MARK; литинг 4, строка 12).

Ограничение коалесинга для ядра операции DELETE (kernel_del_persistent) обусловлено отсутствием возможности барьерной синхронизации всех потоков сетки в модели SIMT, вследствие чего необходимо использовать паттерн «перманентный поток», который, в свою очередь, предполагает перекрестные обращения к элементам массива K.

Перечисленные причины ограничения коалесинга не могут быть ликвидированы без изменения модифицированного алгоритма A-I.

5.5.3.3 Дефицит ресурсов для выполнения потоков

Общее число одновременно выполняющихся блоков ядра зависит в первую очередь от количества необходимых для выполнения одного потока блока регистров, а также объема необходимой для блока разделяемой памяти:

чем меньше необходимое число регистров и необходимый объем разделяемой памяти, тем большее количество блоков может выполняться одновременно [5].

Распределение регистров и разделяемой памяти между блоками происходит во время компиляции.

Как неоднократно упоминалось ранее, по умолчанию для акселератора Tesla K20m используется 64 регистра на один поток, что позволяет выполняться лишь одному блоку максимального размера (1024 потока) на одном потоковом мультипроцессоре [5].

Так как каждое ядро реализации модифицированного алгоритма A-I использует не более 20 регистров мультипроцессора, то при компиляции (nvcc) программы необходимо указать это явно с помощью ключа --maxregcount=20, благодаря чему в два раза увеличится количество одновременно выполняющихся блоков на одном мультипроцессоре вследствие более оптимального распределения регистров между блоками, что, в свою очередь, позволит утилизировать вычислительные ресурсы всех мультипроцессоров максимально эффективно.

5.5.3.4 Дивергенция нитей варпа

Удалось получить ядра с минимальным количеством ветвлений («если-тогда-иначе») в ущерб выразительности исходного кода, но в пользу производительности программы. Такое решение мотивированно тем, что разделение нитей варпа операторами управления потоком выполнения замедляет работу конечной программы в силу особенностей архитектуры CUDA – сначала исполняется одна ветвь одной частью варпа, затем другая ветвь – оставшейся частью варпа, то есть уровень параллелизма уменьшается в два раза. Такое явление принято называть дивергенцией (или расхождением) [5] нитей варпа.

Данные, полученные при профилировании программы, доказывают, что дивергенция нитей варпа практически отсутствует (таблица 6).

Пояснение к таблице 6: эффективность варпов [5] – это отношение числа активных потоков варпа к числу всех потоков варпа (32).

Таблица 6 – Статистика эффективности варпов

Ядро	Минимальная эффективность варпа, %	Максимальная эффективность варпа, %	Средняя эффективность варпа, %
kernel_mul_greedy	4	100	86
kernel_mark_greedy	31	100	92
kernel_del_persistent	65	100	96

6 Оценка реализации

6.1 Реальная временная вычислительная сложность

Был проведен ряд вычислительных экспериментов, результаты которых отражены на рисунках 14 и 15.

Для сравнения на графиках также присутствуют данные, полученные в результате экспериментов с OpenMP-реализаций [3] аналогичного алгоритма (использованы 2 процессора по 12 потоков каждый).

Хотя сравнение CUDA и OpenMP не совсем корректно и дает пространство для дискуссии, такое сравнение имеет место в рамках настоящей работы: реальная пространственная сложность CUDA-реализации превосходит аналогичную сложность OpenMP-реализации в среднем в 7 раз.

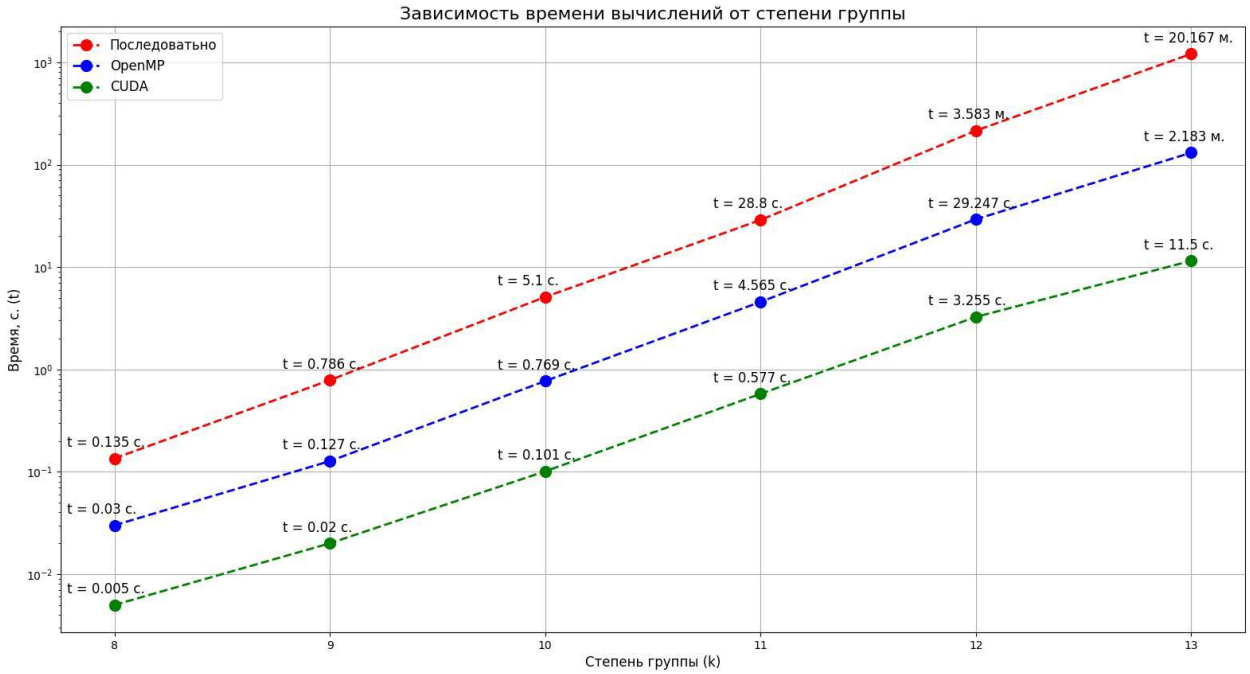

Рисунок 14 – График зависимости времени вычисления от степени группы

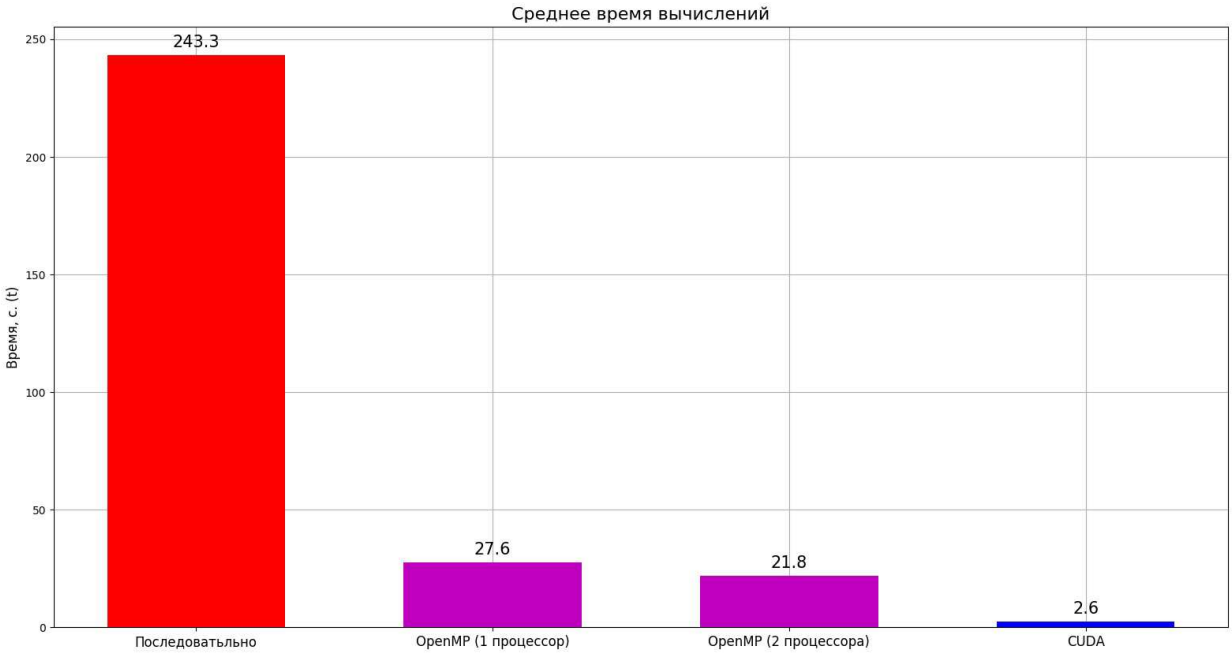

Рисунок 15 – Среднее время вычислений для групп спепеней 8-13

6.2 Реальная пространственная вычислительная сложность

Реальная пространственная сложность полученной CUDA-программы складывается из нескольких компонент, основные из которых: объем глобальной памяти, требуемой для хранения массива K (тип элементов unsigned long long, 8 байтов); объем глобальной памяти, требуемой для хранения массива V (тип элементов unsigned int, 4 байта); объем оперативной памяти, требуемой для хранения массива F (тип элементов unsigned long long, 8 байтов). Использование разделяемой памяти и регистровых файлов мультипроцессоров графического акселератора не учитывается.

Необходимо отметить, что значения представленные посредством графика, который изображен на рисунок 16, являются приблизительными в силу того, что инструменты CUDA SDK не позволяют точно измерить общий объем памяти, требуемый для работы CUDA-программы, кроме того, перед началом вычислений неизвестна необходимая длина массива K, поэтому длина массива K положена равной количеству битов массива V.

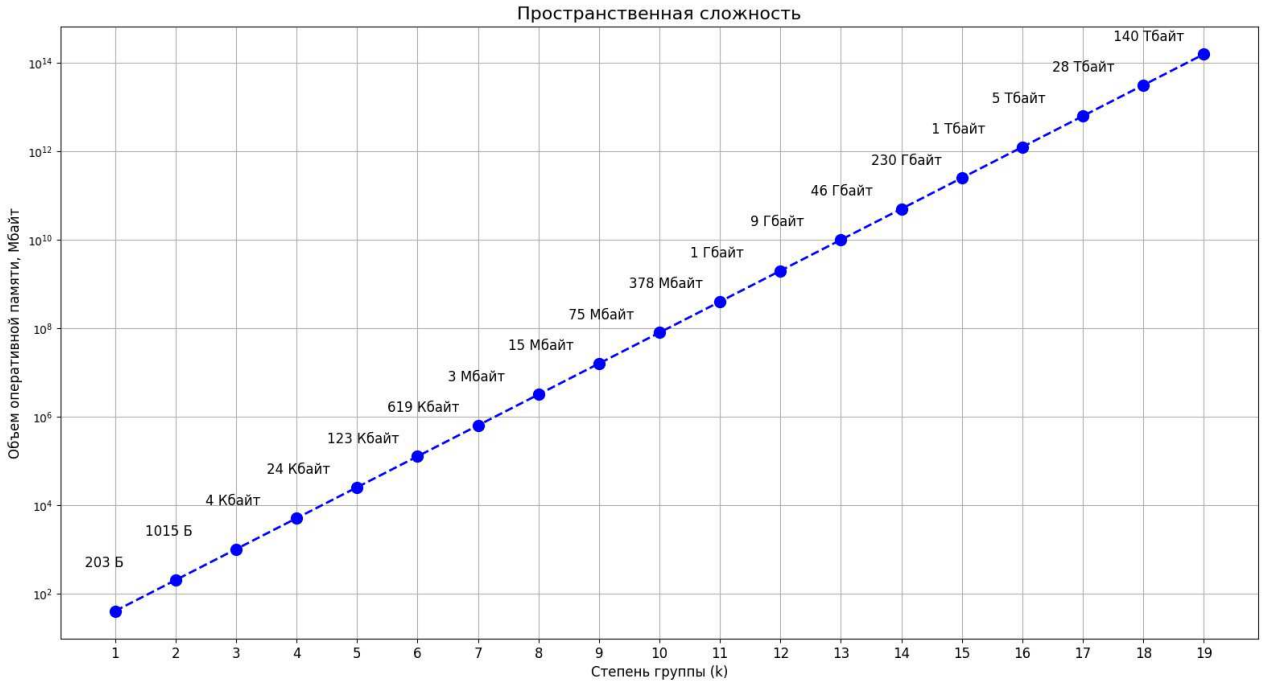

Рисунок 16 – Пространственная сложность реализации

6.3 Масштабируемость

Алгоритмы, обладающие NP-трудностью обычно плохо масштабируются. Это касается и модифицированного алгоритма A-I, а также любой его компьютерной реализации. Не смотря на это, полученная реализация модифицированного алгоритма A-I может успешно работать на вычислительных системах с любым количеством однотипных (с одинаковой архитектурой) графических акселераторов NVIDIA, которые обладают вычислительными способностями выше, чем 3.0, то есть практически на всех современных вычислительных системах с NVIDIA-акселераторами.

Наибольшую проблему для масштабирования представляет собой высокая пространственная сложность модифицированного алгоритма A-I, которая требует насыщенного трафика данных между оперативной памятью компьютера и глобальной памятью графических акселераторов, поэтому эффективность масштабирования напрямую зависит от пропускной способности шины между этими типами памяти.

Для общего случая критерий эффективности масштабирования полученной реализации модифицированного алгоритма A-I можно определить следующим образом: масштабирование реализации модифицированного алгоритма A-I эффективно, если пропускная способность шины, соединяющей оперативную память компьютера и глобальную память графических акселераторов сопоставима со средней пропускной способностью внутренних шин глобальной памяти графических акселераторов.

Так, например, апробация реализации модифицированного алгоритма A-I осуществлялась на аппаратной платформе, где для указанных соединений используется технология PCI Express 2.0 [6] с пиковой пропускной способностью шины 16 Гб/с, что во много раз меньше пиковой пропускной способности шины глобальной памяти акселератора Tesla K20m (208 Гб/с). На основе изложенных фактов, можно сделать вывод о том, что добавление дополнительных акселераторов в слоты PCI Express компьютера, на котором проводилась апробация, не даст прироста производительности программы, а наоборот, снизит ее производительность.

Наиболее эффективное масштабирование можно получить благодаря технологии NVLink [17]. Пиковая пропускная способность шины NVLink значительно превосходит пиковую пропускную способность PCI Express 4.0 [6] – 300 Гб/с против 64 Гб/с. Необходимо обратить внимание на то, что технология NVLink позволяет эффективно соединить как графические ускорители между собой, так и графические ускорители с центральным процессором аналогично PCI Express.

Колоссального прироста производительности программы можно достичь на платформе с архитектурой NVIDIA NVSwitch [18], которая поддерживает до 16 графических ускорителей, соединенных по технологии NVLink. Коммуникация между всеми восемью парами GPU NVIDIA NVSwitch осуществляется со скоростью 300 ГБ/с, причем все 16 соединенных между собой GPU можно использовать как один масштабный ускоритель с унифицированной памятью объемом 0,5 Тб и вычислительной производительностью 2 петафлопса.

ЗАКЛЮЧЕНИЕ

В ходе выполнения настоящей работы поставленные цели были достигнуты, обозначенные задачи выполнены в полной мере – создана первая реализация модифицированного алгоритма A-I для гибридной вычислительной системы, имеющей графические ускорители; проведена ее апробация; дана оценка реализации и выявлены ее преимущества и недостатки.

Хочется надеяться, что данная реализация A-I найдет свое применение в дальнейших исследованиях групп и графов Кэли, а также применение в решении прикладных проблем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Akers, S. group theoretic model for symmetric interconnection networks / S. Akers, B. A. Krishnamurthy // Proceedings of the International Conference on Parallel Processing. – 1986.

2. Schibell, S. Processor interconnection networks and Cayley graphs. Discrete Applied Mathematics / S. Schibell, R. Stafford. – 1992.

3. Кузнецов, А. А. Параллельный алгоритм для исследования графов Кэли групп подстановок / А. А. Кузнецов, А. С. Кузнецова // Вестник СибГУ. –2014. – № 1. – С. 34-38.

4. Holt, D. Handbook of computational group theory / D. Holt, B. Eick, E. O'Brien. – Boca Raton : Chapman & Hall/CRC Press, 2005. – 514 с.

5. NVIDIA developer zone. CUDA Toolkit Documentation [Электронный ресурс]. – Режим доступа: https://docs.NVIDIA.com/cuda/archive/8.0.

6. PCI Express* Architecture [Электронный ресурс]. – Режим доступа: https://www.intel.ru/content/www/ru/ru/io/pci-express.

7. OpenCL Overview [Электронный ресурс]. – Режим доступа: https://www.khronos.org/opencl.

8. Homepage OpenACC. What is OpenACC? [Электронный ресурс]. – Режим доступа: https://www.openacc.org.

9. Top 500. November 2018 [Электронный ресурс]. – Режим доступа: https://www.top500.org/lists/2018/11.

10. Павловская, Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – Санкт-Петербург : Питер, 2012. – 461 с.

11. Скиена, С. С. Алгоритмы. Руководство по разработке / С. С. Скиена – Санкт-Петербург : БВХ-Петербург, 2018. – 720 с.

12. Rauber, T. Parallel programming for Multicore and Cluster systems / T. Rauber, G. Runger – Heidelberg, New York, Dordrecht, London : Springer, 2013. – 516 с.

13. Легалов, А.И. Эволюционное расширение программ при различных парадигмах программирования / А. И. Легалов, И. А. Легалов, А. Ф. Солоха // Труды XVI Байкальской Всероссийской конференции «Информационные и математические технологии в науке и управлении». – 2011. – С 42-49.

14. Боресков, А. В. Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов. – Москва : ДМК Пресс, 2019. – 232 с.

15. Кузнецов, А. А. Быстрое умножение элементов в конечных двупорожденных группах периода 5 / А. А. Кузнецов, А. С. Кузнецова // Вычислительные методы в дискретной математике. – 2013. –№ 1. – С. 110-116.

16. Gupta, K. A study of persistent threads style GPU programming for GPGPU workloads. / K. Gupta, J. A. Stuart, J. D. Owens // Innovative Parallel Computing (InPar) IEEE. – 2012. – С. 1-14.

17. NVIDIA Nvlink [Электронный ресурс]. – Режим доступа: https://www.NVIDIA.com/ru-ru/design-visualization/nvlink-bridges.

18. NVlink Fabric [Электронный ресурс]. – Режим доступа: https://www.NVIDIA.com/ru-ru/data-center/nvlink.

19. СТО 4.2-07-2014 Система менеджмента качества. Общие требования к построению, изложению и оформлению документов учебной деятельности. – Введ. 09-01-2014. – Красноярск: СФУ, 2014. – 60 с.

ПРИЛОЖЕНИЕ А

Исходные тексты CUDA-реализации модифицированного алгоритма A-I

Листинг А.1 – Файл types.h

```
1   #ifndef TYPES_H
2   #define TYPES_H
3
4   #include <stdio.h>
5
6   typedef unsigned int uint32;
7   typedef unsigned long long uint64;
8
9   typedef struct {
10      uint64 full_blocks;
11      uint64 total_blocks;
12      uint64 rem_threads;
13      uint64 total_threads;
14      uint64 rem_threads_offset;
15      uint64 threads_per_block;
16  } kernel_work_conf;
17
18  typedef struct {
19      uint32 omp_threads;
20      uint64 full_work_per_gpu;
21      uint64 rem_work_per_gpu;
22      uint64 rem_threads_offset;
23  } gpus_work_conf;
24
25  typedef enum {
26      PMEM,
27      GMEM,
28      UNDEFINED
29  } memtype;
30
31  typedef struct {
32      uint64 h_s;
33      uint32 *d_V;
34      uint64 *d_K;
35      uint64 *d_P;
36      uint64 *h_F;
37      FILE *logfile;
38      FILE *resfile;
39      size_t msize_F;
40      size_t msize_V;
41      size_t msize_P;
42      size_t msize_K;
43      size_t msize_total;
44      size_t gmem_limit;
45      size_t pmem_limit;
```

```
46          memtype mtype_V;
47          memtype mtype_K;
48          memtype mtype_P;
49          memtype default_memtype;
50          uint32 f_len;
51          float K_size_coef;
52          uint32 dev_num;
53          uint32 default_dev;
54          uint32 logging_enabled;
55          uint32 omp_enabled;
56          uint64 gpu_gmem_total;
57          uint64 gpu_gmem_reserved;
58          uint64 ram_total;
59          uint64 ram_reserved;
60          const char *log_fname;
61          const char *result_fname;
62          cudaError_t status;
63          struct option *options;
64      } app_screen;
65
66      #endif // TYPES_H
```

Листинг А.2 – Файл ai.cuh

```
1       #ifndef AI_CUH
2       #define AI_CUH
3
4       #include <omp.h>
5       #include "config.h"
6       #include "types.h"
7       #include "utils.cuh"
8       #include "kernels.cuh"
9
10      void mul_cuda(uint64 *d_K, uint64 P);
11      void mark_cuda(uint64 *d_K, uint32 *d_V, uint64 P, uint32 gen_degree);
12      void del_cuda(uint64 *d_K, uint64 *d_P, uint64 *h_P, uint32 gen_degree);
13
14      void mul_cuda_omp(uint64 *d_K, uint64 P);
15      void mark_cuda_omp(uint64 *d_K, uint32 *d_V, uint64 P, uint32 gen_degree);
16      void del_cuda_omp(uint64 *d_K, uint64 *d_P, uint64 *h_P, uint32 gen_degree);
17
18      uint64 ai_main_loop_cuda(uint64 *d_K, uint32 *d_V, uint64 *d_P,
19                               uint64 *h_F, uint32 gen_degree);
20
21      uint64 ai_main_loop_cuda_omp(uint64 *d_K, uint32 *d_V, uint64 *d_P,
22                                   uint64 *h_F, uint32 gen_degree);
23
24      #endif // AI_CUH
```

Листинг А.3 – Файл ai.cu

```
1       #include "ai.cuh"
```

```
2
3
4       void mul_cuda(uint64 *d_K, uint64 P)  {
5          static kernel_work_conf kwconf;
6          configure_kernel_work(&kwconf, P);
7          #if GREEDY_KERNELS
8             CALL_CUDA_GREEDY_KERNEL(kernel_mul_greedy, kwconf, d_K, P);
9          #else
10            CALL_CUDA_KERNEL(kernel_mul, kwconf, d_K, P);
11         #endif
12      }
13
14
15      void mark_cuda(uint64 *d_K, uint32 *d_V, uint64 P, uint32 gen_degree) {
16         static kernel_work_conf kwconf;
17         configure_kernel_work(&kwconf, P * gen_degree);
18         #if GREEDY_KERNELS
19            CALL_CUDA_GREEDY_KERNEL(kernel_mark_greedy, kwconf, d_K, d_V);
20         #else
21            CALL_CUDA_KERNEL(kernel_mark, kwconf, d_K, d_V);
22         #endif
23      }
24
25
26      void del_cuda(uint64 *d_K, uint64 *d_P, uint64 *h_P, uint32 gen_degree) {
27         static kernel_work_conf kwconf;
28         configure_kernel_work(&kwconf, (*h_P) * gen_degree);
29         cudaMemset(d_P, 0, sizeof(uint64));
30         #if GREEDY_KERNELS
31            CALL_CUDA_PERSISTENT_KERNEL(kernel_del_persistent, kwconf, d_K, d_P);
32         #else
33                CALL_CUDA_KERNEL_BLOCKS_ORDERED(kernel_del,  kwconf,  d_K,  d_K,
       d_P);
34         #endif
35         cudaMemcpy(h_P, d_P, sizeof(uint64), cudaMemcpyDefault);
36      }
37
38
39      void mul_cuda_omp(uint64 *d_K, uint64 P) {
40         static gpus_work_conf gwconf;
41         static kernel_work_conf kwconf;
42         confugure_gpus_work(&gwconf, P, omp_get_num_threads_from_anywere());
43         configure_kernel_work(&kwconf, gwconf.full_work_per_gpu);
44
45         if(gwconf.full_work_per_gpu) {
46            #pragma omp parallel
47            {
48               cudaSetDevice(omp_get_thread_num());
49               uint64 omp_thread_offset = omp_get_thread_num() \
50                  * gwconf.full_work_per_gpu;
```

```
51
52              CALL_CUDA_KERNEL(kernel_mul, kwconf, d_K + omp_thread_offset, P);
53
54              cudaDeviceSynchronize();
55            }
56          }
57
58          if (gwconf.rem_work_per_gpu) {
59            configure_kernel_work(&kwconf, gwconf.rem_work_per_gpu);
60            CALL_CUDA_KERNEL(kernel_mul, kwconf,
61                        d_K + gwconf.rem_threads_offset, P);
62          }
63        }
64
65
66        void mark_cuda_omp(uint64 *d_K, uint32 *d_V,
67                           uint64 P, uint32 gen_degree) {
68          static gpus_work_conf gwconf;
69          static kernel_work_conf kwconf;
70          uint32 omp_threads_num = omp_get_num_threads_from_anywere();
71
72          confugure_gpus_work(&gwconf, P * gen_degree, omp_threads_num);
73          configure_kernel_work(&kwconf, gwconf.full_work_per_gpu);
74
75          if(gwconf.full_work_per_gpu) {
76            #pragma omp parallel
77            {
78              #pragma omp critical
79              {
80                cudaSetDevice(omp_get_thread_num());
81                uint64 omp_thread_offset = omp_get_thread_num() \
82                   * gwconf.full_work_per_gpu;
83                CALL_CUDA_KERNEL(kernel_mark, kwconf,
84                             d_K + omp_thread_offset, d_V);
85                cudaDeviceSynchronize();
86              }
87            }
88          }
89
90          if (gwconf.rem_work_per_gpu) {
91            configure_kernel_work(&kwconf, gwconf.rem_work_per_gpu);
92            CALL_CUDA_KERNEL(kernel_mark, kwconf,
93                        d_K + gwconf.rem_threads_offset, d_V);
94          }
95        }
96
97
98        void del_cuda_omp(uint64 *d_K, uint64 *d_P,
99                          uint64 *h_P, uint32 gen_degree) {
100         static gpus_work_conf gwconf;
```

```
101     static kernel_work_conf kwconf;
102     uint32 omp_threads_num = omp_get_num_threads_from_anywere();
103
104     confugure_gpus_work(&gwconf, (*h_P) * gen_degree, omp_threads_num);
105     configure_kernel_work(&kwconf, gwconf.full_work_per_gpu);
106     cudaMemset(d_P, 0, sizeof(uint64));
107
108     if(gwconf.full_work_per_gpu) {
109       #pragma omp parallel for ordered
110       for (uint32 i = 0; i < omp_threads_num; i++)
111       {
112         #pragma omp ordered
113         {
114           cudaSetDevice(i);
115           uint64 omp_thread_offset = i * gwconf.full_work_per_gpu;
116           CALL_CUDA_KERNEL_BLOCKS_ORDERED(kernel_del, kwconf,
117                                           d_K + omp_thread_offset,
118                                           d_K, d_P);
119           cudaDeviceSynchronize();
120         }
121       }
122     }
123
124     if (gwconf.rem_work_per_gpu) {
125       CALL_CUDA_KERNEL_BLOCKS_ORDERED(kernel_del, kwconf,
126                                       d_K + gwconf.rem_threads_offset,
127                                       d_K, d_P);
128     }
129
130     cudaMemcpy(h_P, d_P, sizeof(uint64), cudaMemcpyDefault);
131   }
132
133
134   uint64 ai_main_loop_cuda(uint64 *d_K, uint32 *d_V,
135                            uint64 *d_P, uint64 *h_F, uint32 gen_degree) {
136     uint64 h_s, h_P;
137
138     for (h_s = 0, h_P = 1; h_P > 0; h_s++) {
139       h_F[h_s] = h_P;
140       RETURN_IF_CUDA_FAIL(
141         h_s,
142         mul_cuda(d_K, h_P)
143       );
144       RETURN_IF_CUDA_FAIL(
145         h_s,
146         mark_cuda(d_K, d_V, h_P, gen_degree)
147       );
148       RETURN_IF_CUDA_FAIL(
149         h_s,
150         del_cuda(d_K, d_P, &h_P, gen_degree)
```

```
151             );
152         }
153
154         return h_s;
155     }
156
157
158     uint64 ai_main_loop_cuda_omp(uint64 *d_K, uint32 *d_V,
159                                  uint64 *d_P, uint64 *h_F, uint32 gen_degree) {
160         uint64 h_s, h_P;
161
162         for (h_s = 0, h_P = 1; h_P > 0; h_s++) {
163             h_F[h_s] = h_P;
164             RETURN_IF_CUDA_FAIL(
165                 h_s, mul_cuda_omp(d_K, h_P)
166             );
167             RETURN_IF_CUDA_FAIL(
168                 h_s, mark_cuda_omp(d_K, d_V, h_P, gen_degree)
169             );
170             RETURN_IF_CUDA_FAIL(
171                 h_s, del_cuda_omp(d_K, d_P, &h_P, gen_degree)
172             );
173         }
174
175         return h_s;
176     }
```

Листинг А.4 – Файл polynomials.cuh

```
1       #ifndef POLYNOMIALS_CUH
2       #define POLYNOMIALS_CUH
3
4       #include "config.h"
5       #include "types.h"
6       #include "utils.cuh"
7
8       __host__ uint64 get_w(uint32 order);
9       __device__ void b0_product_2gens(uint64 *k1, uint64 *k2);
10
11      #endif // POLYNOMIALS_CUH
```

Листинг А.5 – Файл polynomials.cu

```
1       #include "polynomials.cuh"
2
3
4       __constant__ uint64 w[20] = {
5           1,                  // 5^0
6           5,                  // 5^1
7           25,                 // 5^2
8           125,                // 5^3
9           625,                // 5^4
```

```
10          3125,                   // 5^5
11          15625,                  // 5^6
12          78125,                  // 5^7
13          390625,                 // 5^8
14          1953125,                // 5^9
15          9765625,                // 5^10
16          48828125,               // 5^11
17          244140625,              // 5^12
18          1220703125,             // 5^13
19          6103515625,             // 5^14
20          30517578125,            // 5^15
21          152587890625,           // 5^16
22          762939453125,           // 5^17
23          3814697265625,          // 5^18
24          19073486328125          // 5^19
25      };
26
27      __constant__ int z_power[5][5] = {
28              0, 0, 0, 0, 0,
29              1, 1, 1, 1, 1,
30              1, 2, 4, 3, 1,
31              1, 3, 4, 2, 1,
32              1, 4, 1, 4, 1,
33      };
34
35      __constant__ int z_plus[5][5] = {
36              0, 1, 2, 3, 4,
37              1, 2, 3, 4, 0,
38              2, 3, 4, 0, 1,
39              3, 4, 0, 1, 2,
40              4, 0, 1, 2, 3,
41      };
42
43      __constant__ int z_mult[5][5] = {
44              0, 0, 0, 0, 0,
45              0, 1, 2, 3, 4,
46              0, 2, 4, 1, 3,
47              0, 3, 1, 4, 2,
48              0, 4, 3, 2, 1,
49      };
50
51      __host__ uint64 get_w(uint32 order) {
52          uint64 val;
53          cudaMemcpyFromSymbol(&val, w, sizeof(uint64),
54                               order * sizeof(uint64),
55                               cudaMemcpyDeviceToHost);
56          return val;
57      }
58
59
```

```
60  class Z5 {
61          public:
62                  int number;
63                  Z5 operator=(int a)
64                  {
65                          number = a;
66                          return *this;
67                  }
68  };
69
70  __device__ Z5 operator+(Z5 x, Z5 y) {
71          Z5 Z;
72          Z.number = z_plus[x.number][y.number];
73          return Z;
74  }
75
76  __device__ Z5 operator+(int y, Z5 x) {
77          Z5 Z;
78          Z.number = z_plus[y][x.number];
79          return Z;
80  }
81
82  __device__ Z5 operator+(Z5 y, int k) {
83          Z5 Z;
84          Z.number = z_plus[k][y.number];
85          return Z;
86  }
87
88  __device__ Z5 operator*(Z5 x, Z5 y) {
89          Z5 Z;
90          Z.number = z_mult[x.number][y.number];
91          return Z;
92  }
93
94  __device__ Z5 operator*(int k, Z5 y) {
95          Z5 Z;
96          Z.number = z_mult[k][y.number];
97          return Z;
98  }
99
100 __device__ Z5 pow(Z5 x, int y) {
101         Z5 Z;
102         Z.number = z_power[x.number][y];
103         return Z;
104 }
105
106 __device__ void b0_product_2gens(uint64 *k1, uint64 *k2)
107 {
108
109         Z5 y[ORDER_POWER];
```

```
110     Z5 z[ORDER_POWER];
111
112     uint64 k = (*k1);
113     uint64 k_div_exp;
114
115     #if UNROLL_LOOPS
116        #pragma unroll 15
117     #endif
118     for (uint32 i = 0; i < ORDER_POWER; i++) {
119        k_div_exp = k / EXPONENT;
120        y[i].number = k - k_div_exp * EXPONENT;
121        k = k_div_exp;
122     }
123
124     if ( fast_mod((*k1), 5) < 4 ) {
125        (*k1)++;
126     }
127     else {
128        (*k1) -= 4;
129     }
130
131     z[0] = y[0];
132     #if ORDER_POWER > 1
133        z[1] = (y[1] + 1);
134     #endif
135     #if ORDER_POWER > 2
136        z[2] = (y[0] + y[2]);
137     #endif
138     #if ORDER_POWER > 3
139        z[3] = (2*y[0] + y[3] + 3*pow(y[0],2));
140     #endif
141     #if ORDER_POWER > 4
142        z[4] = (y[4] + y[0]*y[1]);
143     #endif
144     #if ORDER_POWER > 5
145        z[5] = (2*y[0] + y[5] + 2*pow(y[0],2) + pow(y[0],3));
146     #endif
147     #if ORDER_POWER > 6
148        z[6] = (y[6] + 2*y[0]*y[1] + 3*pow(y[0],2)*y[1]);
149     #endif
150     #if ORDER_POWER > 7
151        z[7] = (y[7] + 2*y[0]*y[1] + 3*y[0]*pow(y[1],2));
152     #endif
153     #if ORDER_POWER > 8
154        z[8] = (2*y[0] + y[8] + 2*y[0]*y[1] + 2*y[0]*y[2] + 2*pow(y[0],2)*y[1] \
155           + 3*pow(y[0],2)*y[2] + pow(y[0],3)*y[1] + 2*pow(y[0],2) \
156           + pow(y[0],3));
157     #endif
158     #if ORDER_POWER > 9
159        z[9] = (y[9] + 4*pow(y[0],2)*pow(y[1],2) + 3*y[0]*y[1] \
```

```
160             + y[0]*pow(y[1],2) + 2*pow(y[0],2)*y[1] + 2*y[0]*y[1]*y[2]);
161         #endif
162         #if ORDER_POWER > 10
163             z[10] = (4*y[0] + y[10] + 4*y[0]*y[2] + pow(y[0],3)*y[2] \
164                 + 3*pow(y[0],3) + 3*pow(y[0],4));
165         #endif
166         #if ORDER_POWER > 11
167             z[11] = (3*y[0] + y[11] + pow(y[0],2)*pow(y[1],2) \
168                 + 3*pow(y[0],3)*pow(y[1],2) + 2*y[0]*y[1] + 3*y[0]*y[2] \
169                 + y[0]*pow(y[1],2) + 4*pow(y[0],2)*y[1] \
170                 + 2*pow(y[0],2)*y[2] + 4*pow(y[0],3)*y[1] + 3*pow(y[0],2) \
171                 + 4*pow(y[0],3) + 4*pow(y[0],2)*y[1]*y[2] + 4*y[0]*y[1]*y[2] \
172                 + 2*y[0]*y[1]*y[3]);
173         #endif
174         #if ORDER_POWER > 12
175             z[12] = (y[12] + 2*y[0]*y[1] + 3*pow(y[0],3)*y[1] \
176                 + 3*pow(y[0],2)*y[1]*y[2] + 3*y[0]*y[1]*y[2] + y[0]*y[1]*y[3]);
177         #endif
178         #if ORDER_POWER > 13
179             z[13] = (y[13] + 4*pow(y[0],2)*pow(y[1],2) + 3*pow(y[0],2)*pow(y[1],3) \
180                 + y[0]*y[1] + 2*y[0]*pow(y[1],2) + y[0]*pow(y[1],2)*y[2] \
181                 + 3*y[0]*y[1]*y[2]);
182         #endif
183
184         (*k2) = 0;
185
186         #if UNROLL_LOOPS
187             #pragma unroll 15
188         #endif
189         for (size_t i = 0; i < ORDER_POWER; i++) {
190             (*k2) += z[i].number * w[i];
191         }
192     }
```

Листинг А.6 – Файл kernels.cuh

```
1       #ifndef KERNELS_CUH
2       #define KERNELS_CUH
3
4       #include "types.h"
5       #include "config.h"
6       #include "utils.cuh"
7       #include "polynomials.cuh"
8
9
10      __global__ void kernel_mul(uint64 *K, uint64 P);
11      __global__ void kernel_mark(uint64 *K, uint32 *V);
12      __global__ void kernel_del(uint64 *chkpt_K, uint64 *K, uint64 *P);
13
14      __global__ void kernel_mul_greedy(uint64 *K, uint64 P, uint64 work_size);
15      __global__ void kernel_mark_greedy(uint64 *K, uint32 *V, uint64 work_size);
```

```
16      __global__ void kernel_del_persistent(uint64 *K, uint64 *P, uint64 work_size);
17
18      #endif // KERNELS_CUH
```

Листинг А.7 – Файл kernels.cu

```
1       #include "kernels.cuh"
2
3
4       __global__ void
5         __launch_bounds__(MAX_THREADS_PER_BLOCK)
6         kernel_mul(uint64 *K, uint64 P) {
7
8           uint32 tid = GET_1D_TID();
9
10          b0_product_2gens(K + tid, K + tid + P);
11      }
12
13
14      __global__ void
15        __launch_bounds__(MAX_THREADS_PER_BLOCK)
16        kernel_mark(uint64 *K, uint32 *V) {
17
18          uint32 tid = GET_1D_TID();
19          uint64 k = K[tid];
20          uint64 k_div_32 = k >> 5;
21          uint32 mask_32 = 0x80000000 >> (k - (k_div_32 << 5));
22
23          if (atomicOr(V + k_div_32, mask_32) & mask_32) {
24              K[tid] = 0;
25          }
26      }
27
28
29      __global__ void
30        __launch_bounds__(MAX_THREADS_PER_BLOCK)
31        kernel_del(uint64 *chkpt_K, uint64 *K, uint64 *P) {
32
33          uint64 k = chkpt_K[threadIdx.x];
34          uint32 pos;
35
36                if (k) {
37                  pos = atomicAdd(P, 1);
38                }
39
40          __syncthreads();
41
42          if (k) {
43              K[pos] = k;
44          }
45      }
```

```
46
47
48      __global__ void
49        __launch_bounds__(MAX_THREADS_PER_BLOCK)
50        kernel_mul_greedy(uint64 *K, uint64 P, uint64 work_size) {
51
52          uint32 tid = GET_1D_TID();
53
54          if (tid < work_size) {
55              __shared__ uint64 k[2][MAX_THREADS_PER_BLOCK];
56
57              k[0][threadIdx.x] = K[tid];
58              k[1][threadIdx.x] = K[tid + P];
59
60              b0_product_2gens(k[0] + threadIdx.x, k[1] + threadIdx.x);
61
62              K[tid] = k[0][threadIdx.x];
63              K[tid + P] = k[1][threadIdx.x];
64          }
65      }
66
67
68      __global__ void
69        __launch_bounds__(MAX_THREADS_PER_BLOCK)
70        kernel_mark_greedy(uint64 *K, uint32 *V, uint64 work_size) {
71
72          uint64 tid = GET_1D_TID();
73
74          if (tid < work_size) {
75              uint64 k = K[tid];
76              uint64 k_div_32 = k >> 5;
77              uint32 mask_32 = 0x80000000 >> (k & 0x1F);
78
79              if (atomicOr(V + k_div_32, mask_32) & mask_32) {
80                  K[tid] = 0;
81              }
82          }
83      }
84
85
86      __global__ void
87        __launch_bounds__(MAX_THREADS_PER_BLOCK)
88        kernel_del_persistent(uint64 *K, uint64 *P, uint64 work_size) {
89
90          uint64 tid = GET_1D_TID();
91          uint64 task_id;
92          uint64 pos;
93          uint64 k;
94
95          #if UNROLL_LOOPS
```

```
96              #pragma unroll 512
97          #endif
98          for (uint32 i = 0; i < work_size / blockDim.x + 1; i++) {
99              task_id = tid + i * blockDim.x;
100             if (task_id < work_size) {
101                 k = K[task_id];
102
103                 if (k) {
104                     pos = atomicAdd(P, 1);
105                 }
106
107                 __syncthreads();
108
109                 if (k) {
110                     K[pos] = k;
111                 }
112             }
113             else {
114                 return;
115             }
116         }
117     }
```

Листинг А.8 – Файл utils.cuh

```
1       #ifndef UTILS_CUH
2       #define UTILS_CUH
3
4       #include <stdio.h>
5       #include <time.h>
6       #include <omp.h>
7       #include "config.h"
8       #include "types.h"
9
10      #define GET_1D_TID() \
11          (blockIdx.x * blockDim.x + threadIdx.x)
12
13      #define RETURN_IF_CUDA_FAIL(retval, ...) \
14          __VA_ARGS__; \
15          if (cudaPeekAtLastError() != cudaSuccess) return retval;
16
17      #define CALL_CUDA_KERNEL(kernel, kwconf, mdata_ptr, ...) \
18          if (kwconf.full_blocks > 0) \
19              kernel<<<kwconf.full_blocks, \
20                      kwconf.threads_per_block>>>(mdata_ptr, __VA_ARGS__); \
21          if (kwconf.rem_threads > 0) \
22              kernel<<<1, \
23                      kwconf.rem_threads>>>(mdata_ptr + kwconf.rem_threads_offset, \
24                              __VA_ARGS__);
25
26      #define CALL_CUDA_GREEDY_KERNEL(kernel, kwconf, mdata_ptr, ...) \
```

```
27          if (kwconf.total_blocks > 0) \
28             kernel<<<kwconf.total_blocks, \
29                   kwconf.threads_per_block>>>(mdata_ptr, __VA_ARGS__, \
30                                        kwconf.total_threads);
31
32       #define CALL_CUDA_PERSISTENT_KERNEL(kernel, kwconf, mdata_ptr, ...) \
33          if (kwconf.total_blocks > 0) \
34             kernel<<<1, \
35                   kwconf.threads_per_block>>>(mdata_ptr, __VA_ARGS__, \
36                                        kwconf.total_threads);
37
38       #define CALL_CUDA_KERNEL_BLOCKS_ORDERED(kernel, kwconf, mdata_ptr, ...) \
39          uint64 offset; \
40          for (offset = 0; offset < kwconf.rem_threads_offset; \
41              offset += kwconf.threads_per_block) \
42             kernel<<<1, kwconf.threads_per_block>>>(mdata_ptr + offset, \
43                                              __VA_ARGS__); \
44          if (kwconf.rem_threads > 0) \
45             kernel<<<1, \
46                   kwconf.rem_threads>>>(mdata_ptr + kwconf.rem_threads_offset, \
47                                    __VA_ARGS__);
48
49       #define LOG(file, fmt, ...) \
50          fprintf(file,  "%s " fmt "\n", timestr(), __VA_ARGS__); \
51          fflush(file);
52
53       #define LOG_INFO(file, fmt, ...) \
54          fprintf(file,  "%s [INFO] " fmt "\n", timestr(), __VA_ARGS__); \
55          fflush(file);
56
57       #define LOG_WARNING(file, fmt, ...) \
58          fprintf(file,  "%s [WARNING] " fmt "\n", timestr(), __VA_ARGS__); \
59          fflush(file);
60
61       #define LOG_CRITICAL(file, fmt, ...) \
62          fprintf(file,  "%s [CRITICAL] " fmt "\n", timestr(), __VA_ARGS__); \
63          fflush(file);
64
65       #define EXEC_TIME(elapsed_time, ...) \
66          elapsed_time = omp_get_wtime(); \
67          __VA_ARGS__; \
68          elapsed_time = omp_get_wtime() - elapsed_time;
69
70       #define fast_mod(number, divider) \
71          number - number / divider * divider
72
73       void configure_kernel_work(kernel_work_conf *wconf, uint64 work);
74       void confugure_gpus_work(gpus_work_conf *wconf, uint64 work,
75                           uint32 aviable_omp_threads);
76
```

```
77      memtype any_malloc(void **ptr, uint64 size, memtype default_mtype);
78      memtype any_free(void *ptr, memtype mtype);
79      void any_memset(void *ptr, int val, size_t size, memtype mtype);
80      char *timestr();
81      char *dsize_to_human_view(size_t size);
82      int omp_get_num_threads_from_anywere();
83
84      __global__ void kernel_memset(char *ptr, int val);
85
86      #endif // UTILS_CUH
```

Листинг А.9 – Файл utils.cu

```
1       #include "utils.cuh"
2
3
4       __global__ void
5         __launch_bounds__(MAX_THREADS_PER_BLOCK)
6         kernel_memset(char *ptr, int val) {
7           *(ptr + GET_1D_TID()) = (char) val;
8       }
9
10
11      void configure_kernel_work(kernel_work_conf *wconf, uint64 work) {
12          wconf->threads_per_block = work <= MIN_THREADS_PER_BLOCK ? \
13              MIN_THREADS_PER_BLOCK : MAX_THREADS_PER_BLOCK;
14          wconf->total_threads = work;
15          wconf->full_blocks = wconf->total_threads / wconf->threads_per_block;
16          wconf->rem_threads_offset = wconf->full_blocks \
17              * wconf->threads_per_block;
18          wconf->rem_threads = wconf->total_threads - wconf->rem_threads_offset;
19          wconf->total_blocks = wconf->full_blocks + (wconf->rem_threads ? 1 : 0);
20      }
21
22
23      void confugure_gpus_work(gpus_work_conf *wconf,
24                               uint64 work, uint32 aviable_omp_threads) {
25          wconf->omp_threads = aviable_omp_threads;
26          wconf->full_work_per_gpu = work / wconf->omp_threads;
27          wconf->rem_threads_offset = wconf->omp_threads * wconf->full_work_per_gpu;
28          wconf->rem_work_per_gpu = work - wconf->rem_threads_offset;
29      }
30
31      int omp_get_num_threads_from_anywere() {
32          int omp_threads_num;
33          #pragma omp parallel
34          {
35              #pragma omp single
36              omp_threads_num = omp_get_num_threads();
37          }
38          return omp_threads_num;
```

```
39      }
40
41
42      memtype any_malloc(void **ptr, uint64 size, memtype default_mtype) {
43          if (default_mtype == GMEM) {
44              if (cudaMalloc(ptr, size) == cudaSuccess) {
45                  return GMEM;
46              }
47              cudaGetLastError();
48          }
49          if (cudaMallocHost(ptr, size) == cudaSuccess) {
50              return PMEM;
51          }
52          return UNDEFINED;
53      }
54
55
56      memtype any_free(void *ptr, memtype mtype) {
57          if (mtype == GMEM) {
58              cudaFree(ptr);
59          }
60          else if (mtype == PMEM) {
61              cudaFreeHost(ptr);
62          }
63          return mtype;
64      }
65
66
67      void any_memset(void *ptr, int val, size_t size, memtype mtype) {
68          if (mtype == GMEM) {
69              static kernel_work_conf kwconf;
70              configure_kernel_work(&kwconf, size);
71              CALL_CUDA_KERNEL(kernel_memset, kwconf, (char *) ptr, val);
72          }
73          else if (mtype == PMEM) {
74              memset(ptr, val, size);
75          }
76      }
77
78
79      char *timestr() {
80          time_t current_time = time(NULL);
81          char *time_str = strtok(ctime(&current_time), "\n");
82          return time_str;
83      }
84
85
86      char *dsize_to_human_view(size_t size) {
87          static const char *data_dim_names[4] = {"B", "KB", "MB", "GB"};
88          static char human_view[32];
```

```
89          double human_size = (double) size;
90          uint32 dim;
91          for (dim = 0; (human_size >= 1024) && (dim < 3); dim++) {
92              human_size /= 1024;
93          }
94          sprintf(human_view, "%.1lf%s", human_size, data_dim_names[dim]);
95          return human_view;
96      }
```

Листинг А.10 – Файл main.cuh

```
1       #ifndef MAIN_CUH
2       #define MAIN_CUH
3
4       #include <stdio.h>
5       #include <string.h>
6       #include <getopt.h>
7       #include <omp.h>
8
9       #include "types.h"
10      #include "config.h"
11      #include "utils.cuh"
12      #include "ai.cuh"
13
14      static struct option options[] = {
15          { "f_len", required_argument, NULL, 0 },                // 0
16          { "k_size_coef", required_argument, NULL, 0 },          // 1
17          { "dev_num", required_argument, NULL, 0 },              // 2
18          { "def_dev", required_argument, NULL, 0 },              // 3
19          { "gmem_total", required_argument, NULL, 0 },           // 4
20          { "gmem_reserved", required_argument, NULL, 0 },        // 5
21          { "ram_total", required_argument, NULL, 0 },            // 6
22          { "ram_reserved", required_argument, NULL, 0 },         // 7
23          { "log_fname", required_argument, NULL, 0 },            // 8
24          { "res_fname", required_argument, NULL, 0 },            // 9
25          { "def_memtype", required_argument, NULL, 0 },          // 10
26          { "logging", no_argument, NULL, 0 },                    // 11
27          { "omp", no_argument, NULL, 0 },                        // 12
28          { "help", no_argument, NULL, 0 },                       // 13
29          { 0, 0, 0, 0 }
30      };
31
32      app_screen app_create();
33      void app_terminate(app_screen *app);
34      void app_configure(app_screen *app, int argc, char **argv);
35      void app_open_files(app_screen *app);
36      void app_cuda_init(app_screen *app);
37      void app_malloc(app_screen *app);
38      void app_print_run_info(app_screen *app);
39      void app_run_ai_loop(app_screen *app) ;
40      void app_check_ai_result(app_screen *app);
```

```
41      void parse_args(app_screen *app, int argc, char **argv);
42      void print_help(app_screen *app);
43      int main(int argc, char **argv);
44
45      #endif // MAIN_CUH
```

Листинг А.11 – Файл main.cu

```
1       #include "main.cuh"
2
3
4       app_screen app_create() {
5           app_screen app = {
6               .h_s = 0,
7               .d_V = NULL,
8               .d_K = NULL,
9               .d_P = NULL,
10              .h_F = NULL,
11              .logfile = NULL,
12              .resfile = NULL,
13              .msize_F = 0,
14              .msize_V = 0,
15              .msize_P = 0,
16              .msize_K = 0,
17              .msize_total = 0,
18              .gmem_limit = 0,
19              .pmem_limit = 0,
20              .mtype_V = UNDEFINED,
21              .mtype_K = UNDEFINED,
22              .mtype_P = UNDEFINED,
23              .default_memtype = DEFAULT_MEMTYPE,
24              .f_len = F_MAX_LEN,
25              .K_size_coef = K_SIZE_COEF,
26              .dev_num = DEV_NUMBER,
27              .default_dev = DEV_ID,
28              .logging_enabled = 0,
29              .omp_enabled = 0,
30              .gpu_gmem_total = GPU_GMEM_TOTAL,
31              .gpu_gmem_reserved = GPU_GMEM_RESERVED,
32              .ram_total = RAM_TOTAL,
33              .ram_reserved = RAM_RESERVED,
34              .log_fname = LOG_FNAME,
35              .result_fname = RESULT_FNAME,
36              .status = cudaSuccess,
37              .options = options
38          };
39          return app;
40      }
41
42
43      void app_terminate(app_screen *app) {
```

```
44      if (app->h_F) free(app->h_F);
45      if (app->d_V) any_free(app->d_V, app->mtype_V);
46      if (app->d_K) any_free(app->d_K, app->mtype_K);
47      if (app->d_P) any_free(app->d_P, app->mtype_P);
48      if (app->logfile) fclose(app->logfile);
49      if (app->resfile) fclose(app->resfile);
50      exit(app->status);
51  }
52
53
54  void app_configure(app_screen *app, int argc, char **argv) {
55      parse_args(app, argc, argv);
56
57      app->gmem_limit = app->gpu_gmem_total - app->gpu_gmem_reserved;
58      app->pmem_limit = app->ram_total - app->ram_reserved;
59      app->msize_F = app->f_len * sizeof(uint64);
60
61      uint64 w = get_w(ORDER_POWER);
62      app->msize_V = (w / 32 + 1) * sizeof(uint32);
63      app->msize_K = app->K_size_coef * w;
64      app->msize_P = sizeof(uint64);
65
66      app->msize_total = app->msize_F + app->msize_V + \
67          app->msize_P + app->msize_K;
68  }
69
70
71  void app_open_files(app_screen *app) {
72      app->logfile = freopen(app->log_fname, "w", stderr);
73      if (!app->logfile) {
74          LOG_WARNING(stderr,
75              "Could not open log file \"%s\" (log redirected to CLI)",
76              app->log_fname
77          );
78      }
79      app->resfile = freopen(app->result_fname, "w", stdout);
80      if (!app->resfile) {
81          LOG_WARNING(stderr,
82              "Could not open result file \"%s\" (result redirected to CLI)",
83              app->result_fname
84          );
85      }
86  }
87
88
89  void app_cuda_omp_init(app_screen *app) {
90      int num_gpus;
91      app->status = cudaGetDeviceCount(&num_gpus);
92      if (num_gpus == 0 || app->status != cudaSuccess) {
93          LOG_CRITICAL(stderr, "No CUDA capable devices were detected | %s",
```

```
94              cudaGetErrorString(app->status)
95           );
96           app_terminate(app);
97        }
98
99        if (num_gpus < app->dev_num) {
100          LOG_WARNING(stderr,
101             "The actual number of devices is less than the requested: %d < %d",
102             num_gpus, app->dev_num
103          );
104          app->dev_num = num_gpus;
105       }
106
107       if (num_gpus <= app->default_dev) {
108          LOG_WARNING(stderr,
109             "Unknown device identifier: %d. "
110             "The default (%d) device will be used.",
111             app->default_dev, DEV_ID
112          );
113          app->default_dev = DEV_ID;
114       }
115
116       app->status = cudaSetDevice(app->default_dev);
117       if (app->status != cudaSuccess) {
118          LOG_CRITICAL(stderr, "Could not connect device %d. | %s",
119             app->default_dev, cudaGetErrorString(app->status)
120          );
121          app_terminate(app);
122       }
123
124       if (app->omp_enabled && app->dev_num == 1) {
125          LOG_WARNING(stderr, "%s",
126             "Cannot use OpenMP for single GPU (OpenMP will be disabled)"
127          );
128          app->omp_enabled = !app->omp_enabled;
129       }
130
131       if (!app->omp_enabled && app->dev_num > 1) {
132          LOG_WARNING(stderr, "%s",
133             "Cannot use multiple GPUs without OpenMP (OpenMP will be enabled)"
134          );
135          app->omp_enabled = !app->omp_enabled;
136       }
137
138       if (app->omp_enabled) {
139          if (app->default_memtype == GMEM) {
140             LOG_WARNING(stderr, "%s",
141                "Global memory usage with OMP is not "
142                "supported (pinned memory will be used)"
143             );
```

```
144             app->default_memtype = PMEM;
145         }
146         int num_omp_threads = omp_get_max_threads();
147         if (num_omp_threads < app->dev_num) {
148             LOG_WARNING(stderr,
149                 "The actual number of devices is greater than the number "
150                 "of OMP threads (single thread will be used): %d < %d",
151                 num_omp_threads, app->dev_num
152             );
153             app->omp_enabled = !app->omp_enabled;
154         }
155         else {
156             omp_set_num_threads(app->dev_num);
157         }
158     }
159
160     #pragma omp parallel if(app->omp_enabled)
161     {
162         uint32 tid = omp_get_thread_num();
163         app->status = cudaSetDevice(tid);
164         cudaDeviceReset();
165         #pragma omp critical
166         {
167             if (app->status != cudaSuccess) {
168                 LOG_CRITICAL(stderr, "Could not connect device %d. | %s",
169                     tid, cudaGetErrorString(app->status)
170                 );
171                 app_terminate(app);
172             }
173         }
174         cudaDeviceSynchronize();
175     }
176 }
177
178
179 void app_malloc(app_screen *app) {
180     if (app->msize_total > app->gmem_limit) {
181         LOG_WARNING(stderr, "Memory of GPU limit exceeded: %s",
182             dsize_to_human_view(app->msize_total - app->gmem_limit)
183         );
184     }
185     if (app->msize_total > app->pmem_limit) {
186         LOG_CRITICAL(stderr, "RAM limit exceeded: %s",
187             dsize_to_human_view(app->msize_total - app->pmem_limit)
188         );
189         app_terminate(app);
190     }
191
192     app->h_F = (uint64 *) malloc(app->msize_F);
193     if (!app->h_F) {
```

```
194         app->status = cudaErrorMemoryAllocation;
195         LOG_CRITICAL(stderr, "Could not allocate %s for F | %s",
196             dsize_to_human_view(app->msize_F), cudaGetErrorString(app->status)
197         );
198         app_terminate(app);
199     }
200     else {
201         app->status = cudaSuccess;
202         LOG_INFO(stderr, "%s of RAM are allocated for F | %s",
203             dsize_to_human_view(app->msize_F), cudaGetErrorString(app->status)
204         );
205     }
206
207     app->mtype_V = any_malloc((void **) &app->d_V,
208                         app->msize_V, app->default_memtype);
209     if (app->mtype_V == UNDEFINED) {
210         app->status = cudaGetLastError();
211         LOG_CRITICAL(stderr, "Could not allocate %s for V | %s",
212             dsize_to_human_view(app->msize_V), cudaGetErrorString(app->status)
213         );
214         app_terminate(app);
215     }
216     else if (app->mtype_V == GMEM) {
217         app->status = cudaGetLastError();
218         LOG_INFO(stderr, "%s of global memory are allocated for V | %s",
219             dsize_to_human_view(app->msize_V), cudaGetErrorString(app->status)
220         );
221     }
222     else if (app->mtype_V == PMEM) {
223         app->status = cudaGetLastError();
224         LOG_INFO(stderr, "%s of pinned memory are allocated for V | %s",
225             dsize_to_human_view(app->msize_V), cudaGetErrorString(app->status)
226         );
227     }
228
229     app->mtype_K = any_malloc((void **) &app->d_K,
230                         app->msize_K, app->default_memtype);
231     if (app->mtype_K == UNDEFINED) {
232         app->status = cudaGetLastError();
233         LOG_CRITICAL(stderr, "Could not allocate %s for K | %s",
234             dsize_to_human_view(app->msize_K), cudaGetErrorString(app->status)
235         );
236         app_terminate(app);
237     }
238     else if (app->mtype_K == GMEM) {
239         app->status = cudaGetLastError();
240         LOG_INFO(stderr, "%s of global memory are allocated for K | %s",
241             dsize_to_human_view(app->msize_K), cudaGetErrorString(app->status)
242         );
243     }
```

```
244     else if (app->mtype_K == PMEM) {
245         app->status = cudaGetLastError();
246         LOG_INFO(stderr, "%s of pinned memory are allocated for K | %s",
247             dsize_to_human_view(app->msize_K), cudaGetErrorString(app->status)
248         );
249     }
250
251     app->mtype_P = any_malloc((void **) &app->d_P,
252                               app->msize_P, app->default_memtype);
253     if (app->mtype_P == UNDEFINED) {
254         app->status = cudaGetLastError();
255         LOG_CRITICAL(stderr, "Could not allocate %s for P | %s",
256             dsize_to_human_view(app->msize_P), cudaGetErrorString(app->status)
257         );
258         app_terminate(app);
259     }
260     else if (app->mtype_P == GMEM) {
261         app->status = cudaGetLastError();
262         LOG_INFO(stderr, "%s of global memory are allocated for P | %s",
263             dsize_to_human_view(app->msize_P), cudaGetErrorString(app->status)
264         );
265     }
266     else if (app->mtype_P == PMEM) {
267         app->status = cudaGetLastError();
268         LOG_INFO(stderr, "%s of pinned memory are allocated for P | %s",
269             dsize_to_human_view(app->msize_P), cudaGetErrorString(app->status)
270         );
271     }
272
273     any_memset(app->d_V, 0, app->msize_V, app->mtype_V);
274     app->status = cudaGetLastError();
275
276     if (app->status != cudaSuccess) {
277         LOG_CRITICAL(stderr, "Could not reset V | %s",
278             cudaGetErrorString(app->status)
279         );
280         app_terminate(app);
281     }
282 }
283
284
285 void app_print_run_info(app_screen *app) {
286     LOG_INFO(stderr, "Exponent: %d", EXPONENT);
287     LOG_INFO(stderr, "Order power: %d", ORDER_POWER);
288     LOG_INFO(stderr, "Generators degree: %d", GENERATORS_DEGREE);
289     LOG_INFO(stderr, "Max length of F: %d", app->f_len);
290     LOG_INFO(stderr, "GPUs number: %d", app->dev_num);
291     LOG_INFO(stderr, "Default GPU: %d", app->default_dev);
292     LOG_INFO(stderr, "Max threads per block: %d", MAX_THREADS_PER_BLOCK);
293     LOG_INFO(stderr, "Kernels type: %s",
```

```
294             GREEDY_KERNELS ? "greedy" : "non greedy");
295       LOG_INFO(stderr, "Total global memory of GPU: %s",
296         dsize_to_human_view(app->gpu_gmem_total)
297       );
298       LOG_INFO(stderr, "Reserved memory of GPU: %s",
299         dsize_to_human_view(app->gpu_gmem_reserved)
300       );
301       LOG_INFO(stderr, "Memory of GPU limit: %s",
302         dsize_to_human_view(app->gmem_limit)
303       );
304       LOG_INFO(stderr, "Total RAM: %s",
305         dsize_to_human_view(app->ram_total)
306       );
307       LOG_INFO(stderr, "Reserved RAM: %s",
308         dsize_to_human_view(app->ram_reserved)
309       );
310       LOG_INFO(stderr, "RAM limit: %s",
311         dsize_to_human_view(app->pmem_limit)
312       );
313       LOG_INFO(stderr, "Default memory type: %s",
314         app->default_memtype == GMEM ? "global memory" : "pinned memory"
315       );
316       LOG_INFO(stderr, "Memory required for data: %s",
317         dsize_to_human_view(app->msize_total)
318       );
319       LOG_INFO(stderr, "Logging: %s",
320         app->logging_enabled ? "enabled" : "disabled"
321       );
322       LOG_INFO(stderr, "OpenMP: %s",
323         app->omp_enabled ? "enabled" : "disabled"
324       );
325       if (app->logging_enabled) {
326         LOG_INFO(stderr, "Log file: %s", app->log_fname);
327         LOG_INFO(stderr, "Result file: %s", app->result_fname);
328       }
329     }
330
331
332     void app_run_ai_loop(app_screen *app) {
333       LOG_INFO(stderr, "%s", "Start of target computations");
334       if (app->omp_enabled) {
335         app->h_s = ai_main_loop_cuda_omp(app->d_K, app->d_V, app->d_P, app->h_F,
336           GENERATORS_DEGREE);
337       }
338       else {
339         app->h_s = ai_main_loop_cuda(app->d_K, app->d_V, app->d_P, app->h_F,
340           GENERATORS_DEGREE);
341       }
342       cudaDeviceSynchronize();
343       app->status = cudaGetLastError();
```

```
344 }
345
346
347 void app_check_ai_result(app_screen *app) {
348     if (app->status == cudaSuccess) {
349         LOG_INFO(stderr, "Target computations completed | %s",
350             cudaGetErrorString(app->status)
351         );
352         fprintf(stdout, "|F| = %zu\n", app->h_s);
353     }
354     else {
355         LOG_CRITICAL(stderr, "Target computations interrupted | %s",
356             cudaGetErrorString(app->status)
357         );
358         fprintf(stdout, "|F| >= %zu\n", app->h_s);
359     }
360     for (uint64 i = 0; i < app->h_s; i++) {
361         fprintf(stdout, "%zu %zu\n", i, app->h_F[i]);
362     }
363 }
364
365
366 void parse_args(app_screen *app, int argc, char **argv) {
367     int optidx;
368     static char log_fname[FNAMES_LEN + 1];
369     static char result_fname[FNAMES_LEN + 1];
370
371     while (getopt_long(argc, argv, "", app->options, &optidx) != -1) {
372         switch (optidx) {
373             case 0:
374                 app->f_len = atoi(optarg);
375                 break;
376             case 1:
377                 app->K_size_coef = atof(optarg);
378                 break;
379             case 2:
380                 app->dev_num = atoi(optarg);
381                 break;
382             case 3:
383                 app->default_dev = atoi(optarg);
384                 break;
385             case 4:
386                 app->gpu_gmem_total = atoi(optarg);
387                 break;
388             case 5:
389                 app->gpu_gmem_reserved = atoi(optarg);
390                 break;
391             case 6:
392                 app->ram_total = atoi(optarg);
393                 break;
```

```
394             case 7:
395                 app->ram_reserved = atoi(optarg);
396                 break;
397             case 8:
398                 strcpy(log_fname, optarg);
399                 app->log_fname = (const char*) log_fname;
400                 break;
401             case 9:
402                 strcpy(result_fname, optarg);
403                 app->result_fname = (const char*) result_fname;
404                 break;
405             case 10:
406                 if (!strcmp(optarg, "GMEM") || !strcmp(optarg, "gmem")) {
407                     app->default_memtype = GMEM;
408                 }
409                 else if (!strcmp(optarg, "PMEM") || !strcmp(optarg, "pmem")) {
410                     app->default_memtype = PMEM;
411                 }
412                 else {
413                     printf("Unknow memory type.\n");
414                 }
415                 break;
416             case 11:
417                 app->logging_enabled = 1;
418                 break;
419             case 12:
420                 app->omp_enabled = 1;
421                 break;
422             case 13:
423                 print_help(app);
424                 app->status = cudaSuccess;
425                 app_terminate(app);
426         }
427     }
428 }
429
430
431 void print_help(app_screen *app) {
432     for (struct option *opt = app->options; opt->name != 0; opt++) {
433         printf("--%s\n", opt->name);
434     }
435 }
436
437
438 int main(int argc, char **argv) {
439     double time_diff;
440     app_screen app = app_create();
441     app_configure(&app, argc, argv);
442     if (app.logging_enabled) app_open_files(&app);
443     app_cuda_omp_init(&app);
```

```
444         app_print_run_info(&app);
445         EXEC_TIME(time_diff, app_malloc(&app));
446         LOG_INFO(stderr, "Time spent on memory allocation: %.3lfs", time_diff);
447         EXEC_TIME(time_diff, app_run_ai_loop(&app));
448         LOG_INFO(stderr, "Time spent on target computations: %.3lfs", time_diff);
449         app_check_ai_result(&app);
450         app_terminate(&app);
451     }
```

Листинг А.12 – Файл config.h

```
1       #ifndef CONFIG_H
2       #define CONFIG_H
3
4       #define EXPONENT                    5U
5       #define ORDER_POWER                 12U
6       #define GENERATORS_DEGREE           2U
7       #define F_MAX_LEN                   50U
8       #define DEV_NUMBER                  1U
9       #define DEV_ID                      1U
10      #define LOGGING_ENABLED             0
11      #define RESULT_FNAME                "res"
12      #define LOG_FNAME                   "log"
13      #define FNAMES_LEN                  128U
14      #define K_SIZE_COEF                 1.0F
15      #define DEFAULT_MEMTYPE             GMEM
16      #define MAX_THREADS_PER_BLOCK       1024U
17      #define MIN_THREADS_PER_BLOCK       32U
18      #define GREEDY_KERNELS              1
19      #define UNROLL_LOOPS                0
20      #define GPU_GMEM_TOTAL              4972412928U     // [bytes]
21      #define GPU_GMEM_RESERVED           150000000U      // [bytes]
22      #define RAM_TOTAL                   202640510976U   // [bytes]
23      #define RAM_RESERVED                20264051098U    // [bytes]
24
25      #endif // CONFIG_H
```

Листинг А.13 – Файл Makefile (файл сборки программы GNU Make)

```
1       SHELL = /bin/bash
2
3       NVCC = nvcc
4       TARGET = cuda-app-gmem-pmem
5       INSTALLPATH = ~/bin
6
7       INCLUDEPATH += -I. \
8               -I./types \
9               -I./utils \
10              -I./ai \
11              -I./entry-point \
12              -I./kernels \
13              -I./polynomials
```

```
14
15      CUDAFLAGS += -code=sm_35 \
16              -arch=compute_35 \
17              -Xptxas -dlcm=ca \
18              -use_fast_math \
19              -maxrregcount=32 \
20              -Xptxas -O3
21
22      CFLAGS += $(INCLUDEPATH) \
23              -Xcompiler -fopenmp \
24              -Xcompiler -Ofast \
25              -Xcompiler -funroll-all-loops \
26              -Xcompiler -march=native \
27              -Xcompiler -m64
28
29      .PHONY: all clean install uninstall
30
31
32      all: main.o utils.o polynomials.o kernels.o ai.o
33              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -o $(TARGET) main.o utils.o
        polynomials.o kernels.o ai.o
34
35      main.o: ./config.h ./types/types.h ./entry-point/main.cuh ./entry-point/main.cu
36              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -c ./entry-point/main.cu
37
38      polynomials.o: ./config.h ./types/types.h ./polynomials/polynomials.cuh
        ./polynomials/polynomials.cu
39              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -Xptxas="-v" -dc
        ./polynomials/polynomials.cu
40
41      utils.o: ./config.h ./types/types.h ./utils/utils.cuh ./utils/utils.cu
42              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -c ./utils/utils.cu
43
44      kernels.o: ./config.h ./types/types.h ./kernels/kernels.cuh ./kernels/kernels.cu
45              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -Xptxas="-v" -keep -dc
        ./kernels/kernels.cu
46
47      ai.o: ./config.h ./types/types.h ./ai/ai.cuh ./ai/ai.cu
48              $(NVCC) $(CFLAGS) $(CUDAFLAGS) -c ./ai/ai.cu
49
50      install:
51              mkdir -p $(INSTALLPATH)
52              cp $(TARGET) $(INSTALLPATH)/$(TARGET)
53
54      uninstall:
55              rm $(INSTALLPATH)/$(TARGET)
56
57      clean:
58              rm -rf *.o $(TARGET) kernels.*
```

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт
Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О. В. Непомнящий
подпись инициалы, фамилия
«28» 07 2019 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 Информатика и вычислительная техника
код – наименование направления

Параллельная программа для вычисления функций роста бернсайдовых двупорожденных групп на гибридных вычислительных системах
тема

Руководитель _____ 27.06.19 доцент, канд. техн. наук Д. А. Кузьмин
подпись, дата должность, ученая степень инициалы, фамилия

Выпускник _____ 27.06.19 С. А. Тарасов
подпись, дата инициалы, фамилия

Нормоконтролер _____ доцент, канд. техн. наук В. И. Иванов
подпись, дата должность, ученая степень инициалы, фамилия
27.06.19

Красноярск 2019